SIDE-CHANNEL SIGNAL ANALYSIS FOR SECURING EMBEDDED AND CYBER-PHYSICAL SYSTEMS

A Dissertation
Presented to
The Academic Faculty

By

Haider Adnan Khan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2020

SIDE-CHANNEL SIGNAL ANALYSIS FOR SECURING EMBEDDED AND CYBER-PHYSICAL SYSTEMS

Approved by:

Prof. Alenka Zajic, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Milos Prvulovic, Advisor
School of Computer Science
Georgia Institute of Technology

Prof. Morris B Cohen
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. David V Anderson
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Azadeh Ansari
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Date Approved: October 5, 2020

If I have seen further it is by standing on the shoulders of Giants.

Sir Issac Newton

This thesis is dedicated to my wife Rini and my son Rishan. Without their support, sacrifice, love, and patience, this would not have been possible.

The thesis is also dedicated to my parents, Ma - Begum Mahbuba Rashida and Baba - Haider Anwar Khan. Baba, we miss you. You will always be in our hearts.

ACKNOWLEDGEMENTS

I express my sincere gratitude to my advisors Prof. Alenka Zajić and Prof. Milos Prvulovic, for their continuous support and guidance for my Ph.D. study and related research. Their expertise and feedback were invaluable in formulating the research questions and methodology. I consider myself fortunate for such amazing advisors and mentors.

I am also thankful to the members of my thesis committee: Prof. Morris B Cohen, Prof. David V Anderson, Prof. Azadeh Ansari, and Prof. Alessandro Orso, for their critical reading, insightful comments, and encouragement. Special thanks must go to Prof. Arie Yeredor, for lending me his expertise and intuition to solve my technical problems. I would also like to thank Dr. Daniela Staiculescu for her continued support.

I am grateful to my colleagues at Electromagnetic Measurements in Communications and Computing Lab for their collaborative effort that made a profound impact on my work. I am also thankful to Bangladesh Student Association at Georgia Tech for making my stay in Atlanta memorable.

I deeply thank my parents, Begum Mahbuba Rashida and Haider Anwar Khan, for their endless love and unconditional support throughout my life. I am also grateful to my sisters, Ananya Laboni and Rana Sultana, and my sister-in-law, Nafisa Kamal, for being my sources of inspiration. Finally, I thank Rini and Rishan, my wife and my son. Rini has been my best friend and a great companion and helped me get through this period in the most positive way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis develops methods that leverage electromagnetic (EM) side-channel signals for non-adversarial and non-intrusive monitoring of embedded and cyber-physical systems, and provides techniques for identifying anomalous/malicious program behavior by detecting deviations in EM emanations, and presents a framework for end-to-end basic-block program execution tracking by monitoring the device's EM side-channel signal.

Side-channels cause unintentional information leakage as a side-effect of hardware activity. While attackers have traditionally exploited side-channel analysis for extracting sensitive information from target systems, recent research has utilized side-channels for non-adversarial monitoring of program execution. Such monitoring can be especially useful for securing resource-constrained security-critical embedded systems.

Various approaches have been proposed in the literature to leverage side-channels for anomaly-based intrusion/malware detection, software profiling, program/code execution tracking, instruction execution modeling, etc. The main drawbacks of the existing approaches are that 1) they are coarse-grained and cannot detect tiny deviations caused by stealthy attacks, 2) they do not scale well for monitoring more complex devices (e.g., devices with faster processors and operating systems), and 3) they do not provide end-to-end detailed program execution monitoring/tracking. As such, these approaches can be ineffective in many practical scenarios.

To successfully leverage side-channels for protecting embedded and cyber-physical systems through non-adversarial and non-intrusive monitoring, our research has 1) designed an intrusion detection system that learns a dictionary of reference EM signatures and exploits the dictionary for identifying anomalous/malicious program behavior, 2) designed neural network to model the monitored device's EM side-channel signal and detect stealthy malware activities through deviations in EM emanations, 3) designed a novel framework that performs basic-block program execution tracing by monitoring the device's EM side-channel signal, and 4) demonstrated that even a single instruction deviation in program execution can be detected with high accuracy via EM side-channel signals captured by a readily available measurement device. The work provides a deep understanding of side-channel analysis for program activity monitoring and can be utilized to secure critical embedded systems.

CHAPTER 1

INTRODUCTION

1.1 Motivation

A side-channel is an unintentional communication channel that causes information leakage as a side-effect of executing a legitimate program on the hardware of a computing system [1]. As hardware activity depends on the executed program, the resulting side-channel can reveal information about the program activity. As such, attackers can extract sensitive information by monitoring and analyzing side-channel signals from a target device.

Side-channel attacks pose a serious security threat for many cryptographic implementations. Attackers have exploited analog side-channel signals such as power consumption (i.e., power side-channel) [2], unintentional electromagnetic emanations (i.e., EM side-channel) [3, 4] and even acoustic emissions (i.e., acoustic side-channel) [5] to extract sensitive information from victim systems. While side-channels are traditionally used for cryptanalysis, the information leakage by side-channels can be leveraged for securing systems through non-adversarial program execution monitoring. For instance, power fingerprinting [6] leverages power side-channel signals for integrity assessment of software-defined radios. Likewise, works in [7, 8] use EM side-channel signals for hardware Trojan detection.

Embedded and cyber-physical systems (CPSs) have become ubiquitous and can impact every aspect of our daily lives. Furthermore, embedded

and CPSs are prevalent in many high-assurance systems, including medical devices such as cardiac pacemakers and insulin pumps [9], nuclear power generation, military systems, transportation systems, autonomous and unmanned vehicles, communication satellites, etc. [10]. These embedded devices collect sensor data, perform data processing and real-time analytics, control actuators, and even execute artificial intelligence (AI) tasks. Experts estimate that in 2019, there were more than 26 billion connected embedded devices worldwide [11]. Furthermore, embedded systems are experiencing exponential growth and are expected to be a USD 6.2 trillion market globally by 2025 [12], with more than 75 billion active devices worldwide [11].

Unfortunately, due to security vulnerabilities such as weak security systems, insecure network services, and lack of security updates, many connected embedded devices are exposed to remote attacks that can cause severe physical and financial damages [13]. Attackers have already targeted different embedded and CPSs including industrial control system [14], smart grid system [15], and embedded medical devices [16]. Furthermore, in 2016, the infamous Mirai botnet [17] compromised more than 600,000 embedded devices (mostly IP cameras and routers) worldwide and crippled many high profile web services via a massive distributed denial of service (DDoS) attack. A comprehensive review of attacks on embedded and CPSs can be found in [18].

The security of embedded devices is a serious concern, and there is a growing need for embedded device monitoring. Securing embedded devices can be a challenging task since CPSs are often severely constrained by limited resources, power, and cost. Unfortunately, the state of the art malware detection techniques such as malware signatures [19, 20], sand-

boxing [21, 22], hardware support [23, 24, 25, 26], machine learning [27, 28], and dynamic analysis [29, 30, 31] require substantial computational power. As such, existing security solutions are not feasible due to their overhead to the system. In addition, many CPSs use customized and proprietary software and hardware, and thus, are difficult to update or upgrade. Furthermore, attackers may be able to control the victim device and may disable the internal monitoring system. Therefore, an isolation between the monitored device and the monitoring system is preferable, especially for security-critical high-assurance systems.

A possible solution for these issues is non-adversarial monitoring of embedded and CPSs through side-channel signal analysis. Side-channel based monitoring does not require any resource or infrastructure on, or any modifications to, the monitored system itself, and thus, does not require any update or upgrade of system software or hardware. As such, this approach is especially suitable for monitoring resource-constrained security-critical embedded devices. Furthermore, side-channel-based monitoring is completely external and non-intrusive. Thus, the deployment of such monitoring systems is relatively simple and does not pose any concern for the disruption of services. Additionally, this approach provides isolation between the monitored system and the monitoring system, and thus, protects the monitoring system from any attack on the monitored system.

Due to these advantages, researchers have recently leveraged both power and EM side-channels for non-adversarial hardware/software activity monitoring. For instance, researchers exploited side-channel signal analysis for anomaly-based intrusion/malware detection to protect embedded devices from malicious attacks [32, 33, 34]. However, these approaches are

either coarse-grained detection frameworks which are unable to detect small changes caused by a stealthy malware (e.g., [33]) or require malware signature to achieve a high detection rate (e.g., [32]) or do not scale well with the complexity of the device (e.g., [34]).

In addition to anomaly detection, researchers exploited side-channel signals for zero-overhead software profiling [35], loop-level program activity monitoring [36], and for recognizing a sequence of instructions [37]. Although program profiling is useful for path execution counting, effective monitoring of security-critical devices requires an end-to-end detailed (e.g., basic-block-granularity) program execution tracing. Furthermore, there is very little intuition about the limits of side-channel analysis (e.g., whether a single instruction deviation can be successfully detected), and how these limits are affected by the monitored signal quality (e.g., SNR and bandwidth).

These shortcomings demonstrate that new approaches are required for side-channel-based intrusion/malware detection systems that can detect even stealthy attacks with high accuracy and low detection latency. Furthermore, such detection systems must be effective for monitoring different embedded devices, including devices with faster/complex processors that run on operating systems. Apart from anomaly detection, there is also a dire need for monitoring systems that can leverage side-channel signals for detailed (basic-block granularity) program execution tracing. Such monitoring systems can ensure the embedded system's integrity and security, and would also be useful for embedded device testing, fault diagnostics and debugging, performance analysis and code optimization, and digital forensics. Furthermore, to successfully implement any side-channel-based monitoring system, we must systematically investigate and

understand the empirical/practical limits of side-channel analysis, and evaluate the impacts of signal quality (e.g., signal-to-noise ratio and signal bandwidth) on the monitoring capability.

While both power and EM side-channels can be exploited for program execution monitoring, unlike power, EM side-channel does not require physical access to the system. In addition, EM side-channel often provides higher signal bandwidth. Thus, EM side-channel is preferable in scenarios where (1) physical access to the system is not available (due to packaging), and (2) detailed monitoring of faster devices is required. As such, throughout this thesis, we focus on EM side-channels. The following sections summarize the contributions of the thesis.

1.2 Intrusion Detection through Electromagnetic Signal Analysis

Side-channel analysis is traditionally exploited for unauthorized eavesdropping. However, it is possible to utilize side-channels for securing computing systems through non-adversarial and non-intrusive monitoring. In that regard, researchers have proposed several approaches to monitor hand-held mobile device's power fluctuations to detect computation-intensive malicious activities [38, 39, 40, 41]. For instance, VirusMeter [40] monitored battery usage to identify "long-term" mobile malware. Likewise, [39] correlated power signatures to detect energy-greedy malware. While these approaches can identify computation-intensive attacks that drain the device's battery, they are ineffective against stealthy attacks. Furthermore, Power Fingerprinting (PFP) [6] leveraged power side-channels for integrity assessment of software-defined radios. PFP correlated the processor's power consumption with stored trusted power signatures to identify deviations in execution. However, this approach was

not effective for monitoring more complex devices (e.g., devices with faster processors and operating systems). In contrast, WattsUpDoc [32] used supervised learning (i.e., K-nearest neighbors, multilayer perceptron, and random forests) to identify malicious activity in embedded medical devices. It extracted statistical features along with spectral (i.e., frequency domain) features from real-time dynamic power consumption for malware detection. However, it required training with malware signatures and thus was not effective against zero-day attacks (i.e., new/unknown malware).

In addition, researchers have exploited unintentional electromagnetic emanations (i.e., EM side-channel) for non-intrusive and contactless monitoring of embedded devices. Sehatbakhsh et al. [36] observed that repetitive program activities (e.g., loops) emanate periodic EM emanations and can be detected as spectral peaks or spikes on the EM spectrogram. Spectral Profiling [36] exploited these spectral spikes for "loop-level" program profiling. EDDIE [33] further extended Spectral Profiling to identify tiny intrusions/deviations inside loops using Kolmogorov-Smirnov (KS) test. However, EDDIE could only detect much larger (>500,000 instructions) deviations outside of the loops. In contrast, Han et al. [34] analyzed the EM spectrogram using a long short-term memory (LSTM) network to identify malicious activity on programmable logic controllers (PLCs). However, [34] does not scale well for more complex devices, such as those with faster processors and operating systems.

More importantly, the signal spectrogram is estimated using the short-time Fourier transform (STFT). However, STFT implicitly assumes that the underlying signal is periodic within each fast Fourier transform (FFT) window. While this assumption is reasonable for program loops, it does not hold for non-loop program activities. This non-periodicity often leads to

unintended artifacts in the signal spectrogram. Consequently, EM spectrogram analysis often struggles to identify tiny deviations outside program loops. Furthermore, to identify minute shifts in spectral peaks, spectrogram analysis requires high spectral (i.e., frequency-domain) resolution. However, the higher spectral resolution requires larger FFT windows, which, in turn, leads to higher detection latency.

To address these issues, in Chapter 3, we present IDEA (Intrusion Detection through Electromagnetic signal Analysis), a novel framework that leverages EM side-channel signals for anomaly-based intrusion detection. IDEA is effective against stealthy attacks and can identify tiny deviations in EM side-channel signals, both inside and outside the program loops. For this, IDEA analyzes the monitored device's EM signatures in time-domain (instead of EM spectrogram or frequency domain signatures) and compares them to trusted reference EM signatures. The first step in IDEA is to record EM emanations from an uncompromised reference device to create a baseline dictionary of signal fragments that correspond to the normal behavior of the device. IDEA also exploits clustering to reduce the number of dictionary entries for efficient computing. IDEA then continuously monitors the target device's EM emanations, comparing the observed EM emanations against the baseline dictionary. When no malware is present, the device's EM emanations match the entries in the baseline dictionary well. If, however, the observed EM emanations deviate significantly from the entries in the baseline dictionary, IDEA reports this as an anomaly, which is potentially caused by malware. Finally, we demonstrate with experimental evaluations that IDEA is effective against different malware behavior such as DDoS and Ransomware, and can monitor different devices, including FPGAs, IoTs, and CPSs with high detection accuracy (AUC

>99.5%), from up to 3 m distances.

1.3 Malware Detection using Neural Network Model for Electromagnetic Side-Channel Signals

Information leakage through side-channels is a severe security threat. Interestingly, side-channels can also be leveraged for protecting resource-constrained embedded devices through non-adversarial and non-intrusive hardware/software activity monitoring. In Chapter 3, we have presented IDEA - an intrusion detection framework that exploits the device's EM signatures to identify anomalous device activity. IDEA uses a reference dictionary of trusted EM patterns to verify EM emanations from the monitored device to detect intrusion. Experimental evaluations demonstrated that IDEA can detect stealthy attacks on different embedded systems with excellent accuracy. However, a concern with this approach is its scalability; i.e., the reference dictionary may grow prohibitively large for larger applications. Specifically, computational time and space requirements for IDEA increases linearly with the number of dictionary entries. While IDEA exploits clustering to reduce the number of dictionary entries, for large applications, IDEA can still become computationally inefficient.

To overcome this issue, in Chapter 4, we present a novel malware detection system that leverages neural network models for EM side-channel signals to identify deviations in program execution. A neural network can approximate complicated functions by adjusting its network parameters (i.e., weights and biases) through training. Consequently, a neural network does not require to explicitly store/memorize the reference EM signatures, and thus, removes the necessity for the reference dictionary. While the training phase can be time-consuming, the inference or prediction by a

neural network is straightforward and computationally efficient. For malware detection, we first train a neural network with EM emanations from an uncompromised reference device to model the device's baseline EM signatures. Next, we use this signal model to monitor an unverified target device. Any deviation in the device's activity causes variations in its EM fingerprints, which, in turn, violates the trained model. We identify and report this as an anomalous/malicious device activity. Experimental evaluations reveal that the system can effectively monitor different embedded devices (e.g., FPGAs, IoTs, and CPSs) and detect even stealthy malware attacks (e.g., 5 μs long attack) with excellent detection accuracy (AUC ≈ 0.99) and low detection latency (≈ 20 μs). Furthermore, in our experiments, this system was roughly 35 times faster and 375 times space/memory efficient than that of IDEA.

1.4 Program Tracing through Electromagnetic Side-Channel Signal Analysis

Program tracing is a dynamic program analysis that records the execution path for detailed program activity monitoring. Such traces are commonly used for software testing [42], fault diagnostics and debugging [43], performance analysis and code optimization [44], digital forensics, etc. In addition, program tracking can ensure system integrity and security.

Typically, program tracing is implemented using software instrumentation that records runtime events. However, this adds high (typically 50% [45]) overhead to the program execution. Thus, software instrumentation is not suitable for resource-constrained embedded devices.

An alternative approach is to monitor resource-constrained embedded devices via side-channel analysis. For instance, Spectral Profiling [36] ex-

ploits EM side-channel signals to identify program loops and performs loop-level program profiling. Likewise, zero-overhead profiling [35] correlates EM signals with acyclic program paths to provide execution counts for such paths. Similarly, [46] envisions program hot paths as Markov states and exploits short-time Fourier transform (STFT) of EM signal for program profiling. While loop identification/verification and path execution counts are useful in many applications, effective device monitoring requires end-to-end basic-block-granularity execution tracking. In addition, [37] recovers the most likely executed instruction sequence by modeling the code execution and its power consumption using a hidden Markov model (HMM). Likewise, [47] modeled instruction execution and performed instruction sequence tracking via EM side-channel signal. However, for faster devices, instruction-level modeling requires prohibitively high monitoring bandwidth (e.g., at least a few GHz sampling rate for monitoring a 1 GHz processor). Moreover, single instruction modeling ignores the effects of instruction pipelines. Thus, this approach may not scale well for faster and more complex devices (e.g., devices with operating systems).

To overcome these limitations, in Chapter 5, we present P-TESLA - Program Tracing through Electromagnetic Side-channel Analysis. P-TESLA exploits the device's electromagnetic (EM) emanations to reconstruct a detailed (basic-block-level) program execution path with high accuracy. For this, we use a two-step training process that leverages instrumented training to annotate the uninstrumented training signals and identify which signal snippets correspond to which code segments. Next, we use a novel signal matching technique that efficiently establishes a correspondence between the test signal and the training signals, and use this signal correspondence to reconstruct the program execution path. Finally, we demon-

strate with empirical evaluations that P-TESLA can effectively monitor different embedded devices (e.g., FPGAs and IoTs) and reconstruct program execution paths with high ($\approx$ 99%) accuracy from up to 1 m distance.

1.5 Impacts of Signal Quality on Side-Channel Analysis

Side-channel signals have long been used in cryptanalysis and recently as a means for non-adversarial and non-intrusive hardware/software activity monitoring. Both of these use-cases have seen steady improvement, allowing ever-smaller deviations in program behavior to be monitored (to track program behavior and identify anomalies) or exploited (to steal sensitive information). However, there is very little intuition about the limits of side-channel analysis (e.g., whether a single instruction deviation can be accurately detected), and how these limits are affected by the monitored signal quality (e.g., SNR and bandwidth).

In chapter 6, we use a popular open-source cryptographic software package (an open-source implementation of the RSA public-key cryptosystem) as a test subject to demonstrate that, with enough training data, wide signal bandwidth, and high signal-to-noise ratio, even a single-instruction deviation in program execution can be identified with very high accuracy. We additionally show that, in cryptographic implementations where branch decisions contain information about the secret key, nearly all such information can be extracted from EM side-channel signal. Finally, we analyze how the received signal bandwidth, the amount of training, and the signal-to-noise ratio (SNR) affect the accuracy of side-channel-based reconstruction/prediction of individual branch decisions that occur during program execution.

1.6 Research Contributions

The research contributions of the thesis are:

- *IDEA*, an intrusion detection framework that leverages EM side-channels to protect embedded devices from stealthy attacks [48].
- A malware detection system that exploits neural network to model device's EM side-channel signals to identify anomalous/malicious device activity [49].
- *P-TESLA*, a framework for zero-overhead and non-intrusive program execution tracing through EM side-channel signal analysis [50].
- A systemic investigation of practical limits of EM side-channel analysis, and evaluation of impacts of signal quality on side-channel based program execution monitoring [51].

1.7 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 provides information on side-channels, their use-cases - including traditional side-channel attacks and non-adversarial device monitoring, and topics related to EM side-channel signal analysis. Chapter 3 presents *IDEA* - an intrusion detection system that leverages EM side-channel signals for anomaly detection. Chapter 4 introduces a malware detection system that exploits neural network to identify deviations in EM side-channel signal due to anomalous/malicious device activities. Chapter 5 proposes *P-TESLA* - a framework that exploits EM side-channel analysis for program execution tracing. Chapter 6 empirically evaluates the limits of EM side-channels for

monitoring program execution. Finally, Chapter 7 summarizes the contributions of the thesis and provides possible future research directions.

CHAPTER 2

BACKGROUND

In this chapter, we provide some key concepts related to side-channels, their use cases in attacks and non-adversarial monitoring, and approaches related to side-channel signal monitoring and analysis that were heavily exploited throughout the thesis.

2.1 Side-Channels

Computer systems are carefully designed to protect sensitive information through different security protocols. These protocols ensure that access to any information must be through a proper authorization. However, in 1967, Dr. Willis H. Ware [52] identified vulnerabilities of resource-sharing computer networks that could lead to unintentional information leakage to unauthorized parties. He demonstrated that hardware and software activities in a computer system unintentionally creates undesired signals or "side-channels" that may enable the attackers to bypass the security protocols. In 1973, Lampson [1] exploited these "side-channels" to transmit sensitive customer information to an unauthorized computer program. His work demonstrated the issues related to information confinement in a computing system and proposed a set of safeguards against such information leakage. Furthermore, Eck [53] decoded the electromagnetic (EM) radiations from a video display unit to reconstruct the video content. He claimed that such EM side-channel signals can be picked up and exploited for stealthy eavesdropping, even from distances over 1 kilometer. In addition, Harold [54] noticed that unintentional EM emanations from

processors, communication lines, and output devices caused information leakage.

Moreover, the difference in execution time due to different control-flow branch execution (e.g., conditional statements) can also cause information leakage. Kocher [55] demonstrated such timing attacks to break different cryptosystems (e.g., RSA and DSS). In addition, Schindler [56] exploited timing attacks to compromise the secret keys of RSA with the Chinese remainder theorem. However, these attacks were performed on simple computing devices and required direct/physical access to the system. Bonech et al. [57] demonstrated remote timing attacks that broke secret keys of OpenSSL, a cryptosystem commonly used in web servers and SSL applications, without any direct access to the system.

Furthermore, Kocher [2] proposed differential power analysis (DPA) that exploited power side-channel signals for cryptanalysis. He demonstrated the effectiveness of DPA by breaking various cryptosystems' secret keys on microcontrollers. He collected thousands of power traces to build a statistical model to estimate the secret bits of cryptosystems. He claimed that unanticipated security faults could lead to catastrophic consequences if hardware designers failed or ignored to address the information leakage through side-channels. Moreover, Goubin et al. [58] demonstrated that DPA could circumvent the countermeasures against simple power analysis (SPA) attacks. In addition, Messerges et al. [59] extracted secret exponent bits from tamper-resistant hardware using DPA. To counteract these attacks, researchers proposed several countermeasures. For instance, Chari et al. [60] added randomness while performing key-dependent computations to prevent statistical attacks such as DPA. Likewise, Bayrak et al. [61] identified and transformed sensitive instructions to minimize infor-

mation leakage.

While side-channel attacks often targeted simple devices (e.g., smart cards, simple embedded devices, etc.), researchers also demonstrated attacks on more complex devices, such as smartphones [62, 4] and PCs [3, 63]. For instance, Genkin et al. monitored EM emanations from PCs using a readily-available consumer-grade radio receiver and broke RSA encryption [3]. Likewise, Alam et al. [4] retrieved secret keys of RSA from mobile phones using only a single EM measurement.

Moreover, even acoustic emissions and heat radiations or system temperatures can unintentionally reveal sensitive information. Researchers demonstrated that English words and texts can be reconstructed using acoustic emanations from keyboards [64] and printers [65]. Similarly, work in [5, 66] used acoustic side-channels to extract secret encryption keys. In addition, temperature attacks, such as in [67, 68] correlated circuit activities with heat radiation. Furthermore, Genkin et al. [69] exploited variations in chasis potential to break RSA cryptosystem.

Apart from analog/physical side-channels, attackers have also exploited digital/micro-architectural variations in computer systems. For instance, cache attacks monitor the victims' cache access patterns in a shared physical system (e.g., in a virtualized environment) to extract secret information such as encryption keys [70, 71, 72, 73].

Throughout this thesis, we focused on EM side-channels due to their capability for non-intrusive, detailed, and remote/contact-less program execution monitoring. In the next section, we briefly review EM side-channels and their use cases.

2.2 EM Side-Channels: Attacks and Non-Adversarial Use Cases

In 1985, Wim van Eck [53] published the first unclassified technical analysis of the security risks of EM emanations from electronic devices. Van Eck eavesdropped on a computer system by picking up its monitor's EM emanations and successfully reconstructed the video content at a range of hundreds of meters, using just $15 worth of equipment. His work demonstrated that side-channel EM emanations are present in electronic devices (e.g., keyboards, computer displays, printers, etc.) and can be captured using proper equipment. This opened a new era for EM side-channel attacks. Furthermore, [74] Gandolfi et al. revealed secret bits of RSA and DES cryptosystems by exploiting EM side-channel signals from CMOS chips and smart cards. They correlated the EM signals with the secret keys and demonstrated that EM analysis can be as effective as power analysis. Moreover, Agrawal et al. [75] showed that EM emanations are consequences of current flows within circuits and electronic devices and can cause information leakage due to the correlation between the processed data and current flow. They also demonstrated that EM side-channel attacks can circumvent power analysis countermeasures to break cryptosystems. More recently, Alam et al. [4] retrieved secret keys of OpenSSL's fixed-window constant-time blinded RSA from mobile phones using EM side-channel signal. They first identified the vulnerable part of the program (i.e., the secret bit dependent computations) in the EM signal, and then compared the signal patterns with the known training signals to predict the secret bits with only a single measurement.

While EM side-channel analysis is traditionally exploited for unauthorized eavesdropping, it is possible to utilize side-channels for securing

computing systems through non-adversarial monitoring. In that regard, researchers have used EM side-channels for software profiling, malware detection, program/code execution tracking, instruction modeling, etc.

Callan et al. [35] exploited EM side-channel signals for zero-overhead program profiling that required no hardware or software modification or instrumentation. Zero Overhead Profiling (ZOP) [35] performed a depth-first search (DFS) through the program's control-flow graph (CFG) using EM signatures to identify and count executions of acyclic control-flow paths. Additionally, ZoP or similar graph-search-based profiling requires extensive code coverage (i.e., a thorough set of inputs that execute all relevant code segments or control-flow sub-paths in the training phase). Thus, Rutledge et al. [76] proposed progressive symbolic execution (PSE) to address the code coverage issues to further improve ZoP. In contrast, Sehatbakhsh et al. [36] observed that repetitive program activity (e.g., loops) emanates periodic EM signals and can be detected as spectral peaks or spikes on the spectrogram of the signal. Spectral Profiling [36] exploited these spectral spikes for loop identification and performed "loop-level" program profiling. Similarly, [46] modeled program hot paths, such as loops, as Markov states, and extracted short-time Fourier transform (STFT) features from EM side-channel signals to perform program profiling. Moreover, Yilmaz et al. [47] modeled instruction execution and performed instruction sequence tracking via EM side-channel signal. Work in [77] provides a detailed review of side-channel-based code/program execution monitoring systems.

EDDIE [33] further extended Spectral Profiling [36] to identify tiny intrusions/deviations inside loops. EDDIE noticed that any deviation in program loops causes a spectral shift in the spectrogram and detected such

spectral shifts using the Kolmogorov-Smirnov (KS) test. Similarly, Syndrome [78] exploited EM spectral analysis to protect embedded medical devices against malicious attacks. Likewise, Han et al. [34] analyzed the EM spectrogram using a long short-term memory (LSTM) network to identify malicious activity on programmable logic controllers (PLCs). Furthermore, REMOTE [79] demonstrated a malware detection system that was robust to signal variations due to device-to-device variability such as shift and drift in the device's clock frequency. A detailed review of side-channel based anomaly detection can be found in [80].

Furthermore, EMPROF [81] exploited EM emanations to detect last-level cache (LLC) misses for memory profiling without any instrumentation. EMPROF noticed that the processor stalls associated with the LLC misses cause dips in the EM amplitude and identified these "dips" using moving average filters. In addition, EMMA [82] leveraged electromagnetic side-channel signals for embedded device attestations.

Apart from program execution monitoring, EM side-channels have been exploited for hardware Trojan detection [7, 8] and for enhanced physical authentication [83]. Soll et al. [7] used localized EM measurements to distinguish between malicious and genuine circuit designs on FPGAs. Similarly, Balasch et al. exploited EM fingerprints to identify hardware Trojans in FPGAs. In contrast, Nguyen et al. created a 'backscattering side-channel' by transmitting high-frequency EM signals toward the IC and monitored the backscattered (reflected) signal from the IC [84]. This approach for non-destructive IC inspection using EM signals can be effective for hardware Trojan detection [85, 86] and counterfeit IC detection [87]. In addition, Sakiyama et al. [83] demonstrated that EM emanations from an FPGA device can be exploited as a physically unclonable function

(PUF) for enhanced device authentication.

2.3 Electromagnetic Side-Channel Analysis

To efficiently analyze EM side-channels, we must first understand how electromagnetic emanations are created by hardware activities and identify program execution dependent variations in emanated EM signals.

2.3.1 Electromagnetic Emanations from Hardware Activity

All electronics generate unintentional EM emanations, which EMI/EMC treats as noise. While it may not be immediately obvious, the unintentional EM emanations created by computing devices are intrinsically linked to both the input data and operations (instructions) used during a computation. Researchers have identified [88] and quantified [89] instruction-dependent EM emanations, and localized [90] the sources of these emanations. At each processor cycle, the CPU draws a current that is a direct result of the instruction(s) being executed. Much of this instruction-dependent current is drawn by the CPU clock circuitry and by the circuitry which performs new computations (i.e., switches on and off). This creates a strong current at the CPU clock frequency and acts as a carrier modulated by the clock-to-cycle variations in program activity (i.e., executed instructions). As this carrier modulated current flows through the wires within the processor and on the device's printed circuit board (PCB), it creates EM emanations at the CPU clock frequency (and its harmonics) [91]. This is demonstrated in Figure 2.1 where different program activities were exploited to create different AM sideband signals at different frequencies (f_{alt}) [88]. To analyze such program-related sideband signals, we first amplitude demodulate the monitored EM signal at the CPU clock frequency.

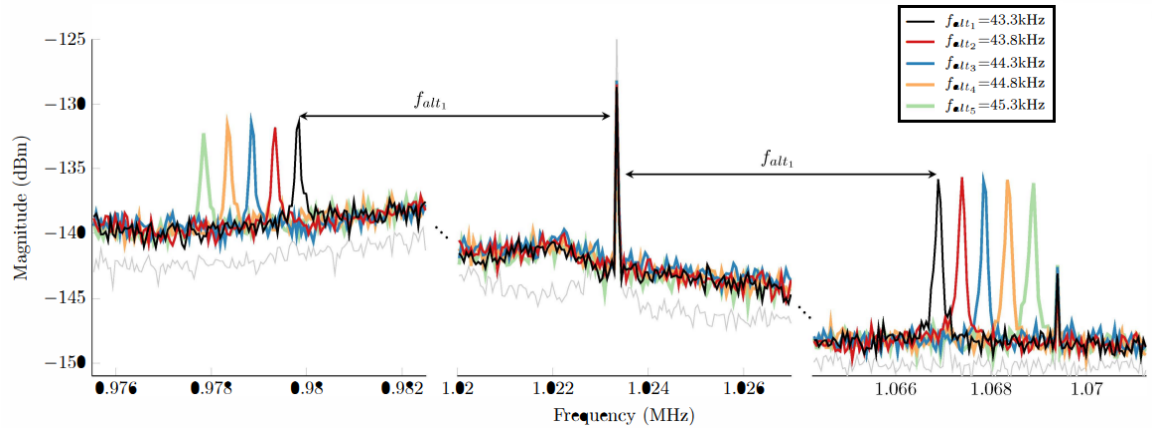


Figure 2.1: A carrier at clock frequency f_c and its left and right sidebands generated by program activity [88].

2.3.2 Amplitude Demodulation of EM side-channel signal

We receive the emanated EM signal using a probe or an antenna, amplitude demodulate the received signal $r(t)$ at the CPU clock frequency f_c, and digitize the demodulated signal using an analog-to-digital converter (ADC).

$$x_a(t) = |r(t) \times e^{j2\pi f_c t}| \tag{2.1}$$

Here, $x_a(t)$ is the amplitude demodulated analog signal, and t denotes the time. The demodulated signal $x_a(t)$ is then passed through an anti-aliasing filter with bandwidth B, and sampled at a sampling period T_s.

$$x_d(n) = x_a(nT_s) \tag{2.2}$$

Here, $x_d(n)$ denotes the sampled signal at sample index n. The anti-aliasing filter cancels unwanted signals with frequencies beyond $f_c \pm B$. Note that, the sampling period T_s is determined by the well known Nyquist criterion:

$$\frac{1}{T_s} > 2B \tag{2.3}$$

CHAPTER 3
INTRUSION DETECTION THROUGH ELECTROMAGNETIC SIGNAL ANALYSIS

3.1 Overview

In this chapter, we present a novel framework called IDEA (Intrusion Detection through Electromagnetic signal Analysis) that exploits electromagnetic (EM) side-channel signals to detect malicious activity on embedded and cyber-physical systems (CPSs). IDEA first records EM emanations from an uncompromised reference device to establish a baseline of reference EM patterns. IDEA then monitors the target device's EM emanations. When the observed EM emanations deviate from the reference patterns, IDEA reports this as an anomalous or malicious activity.

IDEA does not require any resource or infrastructure on, or any modification to, the monitored system itself. As such, IDEA is especially suitable for monitoring resource-constrained security-critical embedded devices. In addition, IDEA is isolated from the target device and monitors the device without any physical contact. This isolation protects IDEA form any attack on the monitored device and ensures that the integrity of IDEA is not compromised even if the monitored device itself is completely compromised. Furthermore, the deployment of IDEA is relatively simple; IDEA does not make any change to the monitored system and thus creates no regulatory, safety, or disruption concern for the system. Moreover, IDEA identifies malicious activities using the trusted references only, without any knowledge of malware signatures. This means that no training on malware or

anomalous behavior is needed and consequently removes the need for regular updates for new malware signatures. Thus, IDEA ensures protection against zero-day attacks.

We evaluate IDEA by monitoring the target device while it is executing embedded applications with malicious code injections such as DDoS, Ransomware, and code modification. We further implement a control-flow hijack attack, an advanced persistent threat, and a firmware modification on three CPSs: an embedded medical device called SyringePump, an industrial PID Controller, and a Robotic Arm, using a popular embedded system, Arduino UNO. The results demonstrate that IDEA can detect different attacks with excellent accuracy (AUC > 99.5%, and 100% detection with less than 1% false positives) from distances up to 3 m.

The main contributions of this chapter are:

- IDEA - a novel framework that leverages the device's EM side-channel signal for intrusion detection.
- A training method that (1) constructs a dictionary of reference EM signatures from an uncompromised reference device, and (2) exploits clustering to reduce the dictionary size for efficient computing.
- An anomaly detection algorithm that (1) reconstructs EM signals using the reference dictionary, and (2) identifies anomalous EM signals based on the reconstruction error.
- Empirical evaluations that demonstrate that (1) IDEA is effective against different malware behavior (e.g., DDoS, Ransomware, etc.) (2) IDEA is effective for monitoring different embedded devices (e.g., FPGAs, IoTs, CPSs), (3) IDEA is robust against noise and interference, and (4) IDEA can detect malicious activities from up to 3 m distance.

The rest of the chapter is organized as follows: Section 3.2 states the envisioned threat model, Section 3.3 describes IDEA - our framework for anomaly-based intrusion detection, Section 3.4 presents the experimental evaluations, and finally, Section 3.5 presents the summary.

3.2 Threat Model

IDEA is an external monitoring system for high-assurance CPSs (such as embedded medical devices) and can detect the execution of malware through the EM side-channel of the target device. The envisioned threat model involves the following assumptions:

1. The monitoring framework (IDEA) does not know the nature of the attack or its EM signature(s) and only relies on the signatures for the monitored application itself. We assume that IDEA always has correct reference models for malware-free signatures of the monitored applications and these models can not be compromised.

2. The adversary has physical and/or remote access to the target device and has prior knowledge of the device and its software. The attacker can thus exploit any vulnerability (e.g., a buffer-overflow) to execute a malicious activity on the system by either launching a separate thread/process and starting a potential cyber-attack (e.g., DDoS) or modifying/re-using the existing application to disrupt or change the original functionality of the targeted system (e.g., control-flow hijack). Furthermore, the adversary can even modify the system's source code and libraries and/or reprogram the system to start a malicious activity. However, IDEA does not know anything about the nature of the attack and only reports an error if an anomaly is detected.

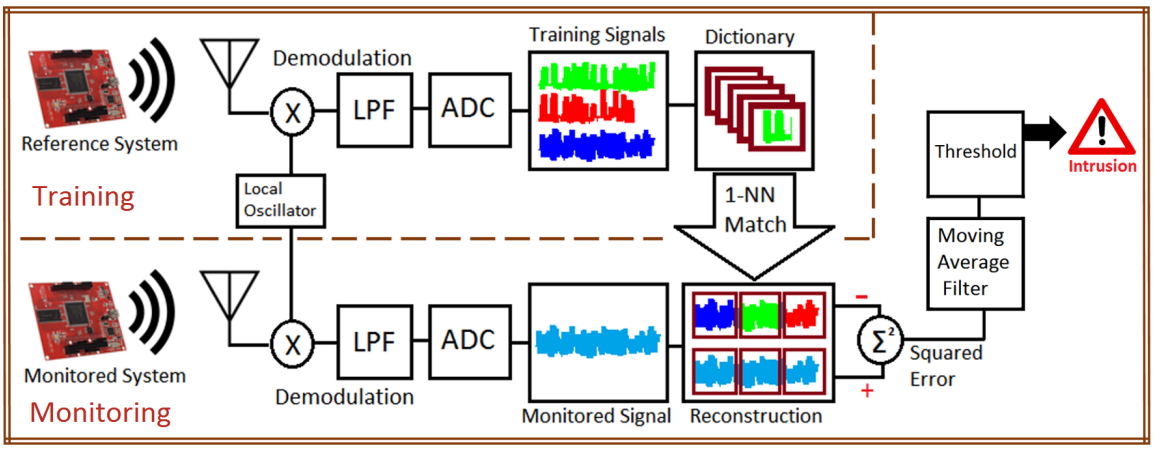


Figure 3.1: Overview of IDEA framework.

3.3 Intrusion Detection through Electromagnetic Signal Analysis

Figure 3.1 illustrates the workflow of IDEA. The signals in both training and monitoring phases are demodulated, low-pass filtered, and sampled before they are subjected to the main part of IDEA signal processing. IDEA exploits techniques similar to template-based pattern matching to identify anomalous (hence, potentially malicious) activity during the program execution. In the training phase, IDEA learns a dictionary of reference EM signatures or "words" by executing trusted programs on an uncompromised reference device. Next, in the monitoring phase, it continuously monitors the target device's EM signal by matching it and reconstructing it using the dictionary. When the reconstruction error is above a predefined threshold (i.e., there is a significant deviation from the reference EM signatures), IDEA reports an anomaly (intrusion). The rest of this section describes IDEA in further detail.

3.3.1 AM Demodulation

Unintentional EM emanations occur at various frequencies, but of particular importance is the frequency band centered around the clock frequency

of the processor, a.k.a. the Central Processing Unit (CPU). This is because this frequency band contains signals that are primarily a function of the instruction sequence executed by the CPU. Each processor cycle, the CPU draws a current, which is a direct result of the instruction(s) execution. Much of this instruction-dependent current is drawn by the CPU clock circuitry and by the circuitry that performs new computations (i.e., switches on and off) every CPU clock cycle. This creates a strong current at the CPU clock frequency, which acts as a carrier modulated by the clock-to-cycle variations in program activity (i.e., executed instructions). These currents flow through the wires within the processor and on the device's printed circuit board (PCB). At CPU and memory clock frequencies (and their harmonics), the EM emanations created can propagate far enough to be observed with a high signal-to-noise ratio [91]. When observed this way, the emanating device has much in common with a communications system since the device is a transmitter which (inefficiently and unintentionally) transmits a message signal carrying information about program activity using an amplitude modulated carrier (i.e., the clock signal). We can then receive and demodulate this signal using wireless communications techniques. All EM signals, both in the training phase and in the monitoring phase, are AM demodulated at the processor clock frequency, low-pass filtered with an anti-aliasing filter, and sampled before being sent for signal processing.

3.3.2 Training Phase: Dictionary Learning

The training phase consists of learning a dictionary of EM signatures through the execution of trusted programs on a reference device. We execute trusted programs on an uncompromised device and record the

corresponding EM signals. We use different inputs to execute different control flow paths, as described in [35]. Ideally, we would like to observe all possible control flow paths. However, in a practical scenario, this may require too many inputs (hence, too many training examples). For example, a twenty level nested IF ELSE condition will have $2^{20} = 1048576$ different execution paths. Nevertheless, we aim at observing most control flow execution paths and try to ensure that even if there are unobserved control flow paths, they are either highly unlikely or relatively brief. These goals are the same as those that guide program testing. So, program inputs created to provide good test coverage of a program are likely to satisfy the needs of IDEA training. Once we have the training signals, we learn the dictionary words using the following process.

Learning Words

The demodulated EM signal is split into multiple overlapping short-duration windows that are recorded as dictionary entries or "words". These words correspond to the EM signature of the underlying program execution. All dictionary words have the same word-length l. Each word is shifted by s samples from the previous one. When s is small, we end up with densely overlapping words. Consequently, we learn a dictionary with a large number of words with slight variability (i.e., shift). This can help to achieve shift-invariant pattern matching. Shift-invariance is necessary because of hardware events, such as cache misses, that delay (or shift) the subsequent execution (and the corresponding EM signal). A cache miss can potentially occur at many different points of the program execution. It is neither practical nor even possible to generate a training set with all possible scenarios of cache hits or misses. Therefore, creating a dictionary with

densely overlapped words can help us match EM signatures better under variability due to hardware activity.

Word Normalization

All dictionary words are post processed by mean subtraction and scale normalization:

$$\mathbf{w} = (\mathbf{w_0} - \mu)/\sigma \tag{3.1}$$

Here, w denotes the normalized word, μ is the mean, and σ is the standard deviation of the unnormalized word $\mathbf{w_0}$.

This normalization improves the matching accuracy by ensuring that the matching is based on the pattern (i.e., the relative shape of the waveform) rather than on the actual amplitude. For instance, due to different positions of the antenna, the distance from the processor may change between the training and the monitoring phase. Hence, the training and the monitored signals can have different scales or amplifications. This normalization nullifies such issues.

Dictionary Reduction through Clustering

Next, we apply clustering to reduce the number of dictionary entries. All applications have loops, which tend to generate repetitive EM patterns. Likewise, the same control flow paths are often reiterated at different points of the execution and generate similar EM patterns. Consequently, the reference dictionary can have a large number of words or patterns that are very similar and correspond to the same code execution. The objective of clustering is to assign similar words or EM patterns into a single cluster, and exploit the cluster centroid as the representative of the cluster. Using cluster centroids as dictionary words improves computational efficiency by

reducing the number of dictionary entries.

As the number of clusters k (i.e., the number of unique EM patterns) is not known a priori, popular clustering algorithms such as k-means can not be used for the dictionary reduction. Instead, we use a threshold-based clustering, where the threshold t is used as a parameter. Given a cluster centroid c_i, the algorithm proceeds by alternating between two steps:

Assignment Step: Assign each unassigned word w_p whose Euclidean distance from the centroid c_i is less than the threshold t to the cluster S_i.

$$\mathbf{S_i} = \{w_p : \|w_p - c_i\|^2 < t \ \wedge \ w_p \notin S_j \ \forall j, \ 1 \leq j < i\} \tag{3.2}$$

Update Step: Update the cluster centroid c_i by averaging all members of cluster S_i.

$$\mathbf{c_i} = \frac{1}{|S_i|} \sum_{w_p \in S_i} w_p \tag{3.3}$$

Once the assignments no longer change, select a new cluster centroid randomly from the words that have not yet been assigned to any cluster. The algorithm converges when all words are assigned to a cluster.

3.3.3 Monitoring Phase: Intrusion Detection

In the monitoring phase, the EM signal is continuously monitored and matched against the dictionary, and anomalous activity is reported when the monitored signal deviates significantly from its dictionary-based reconstruction.

Matching and Reconstruction

The monitored EM signal is split into windows and matched against the dictionary using the 1-Nearest Neighbor algorithm [92], with Euclidean

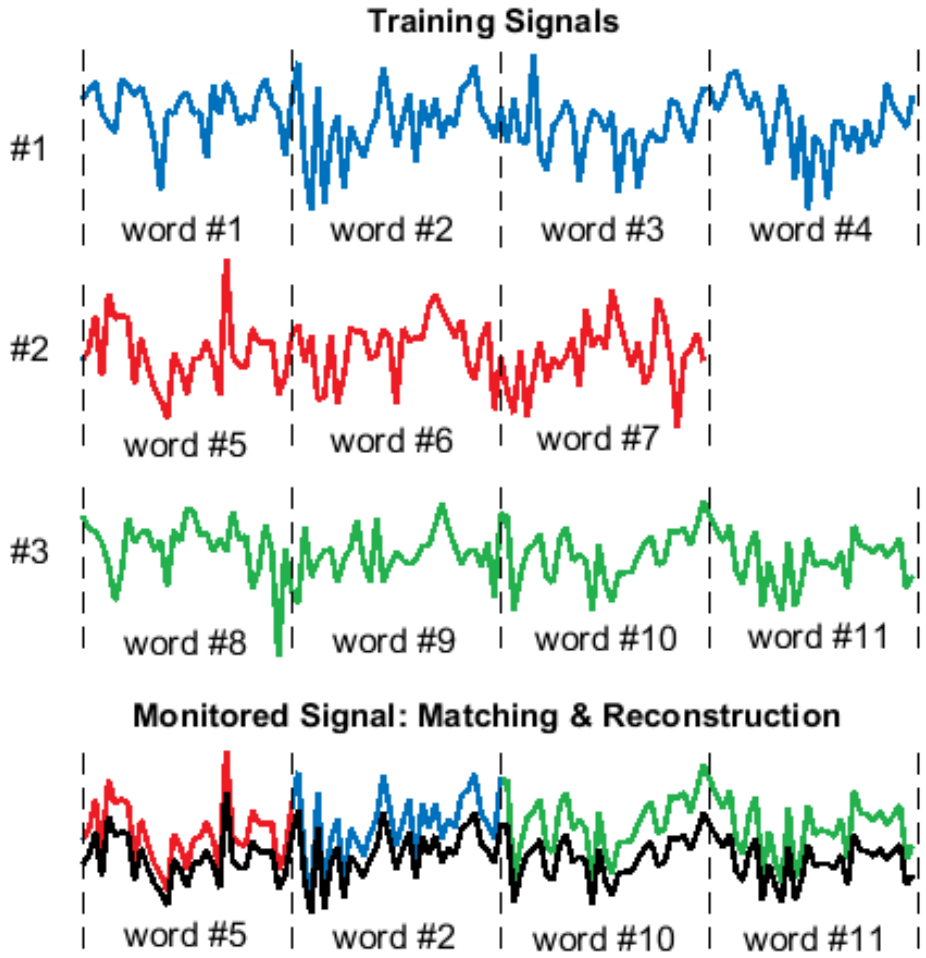


Figure 3.2: An example of verifying the monitored signal using dictionary words. The monitored signal (shown with black lines) is verified against training (red, blue, and green) signals.

distance as the distance metric. The signal is then reconstructed by replacing each window with its best-match dictionary word. This is illustrated in Figure 3.2. The entire signal can be reconstructed by concatenating these best-match words that correspond to the signal's sequence of windows, and this reconstructed signal is then used for anomaly (intrusion) detection.

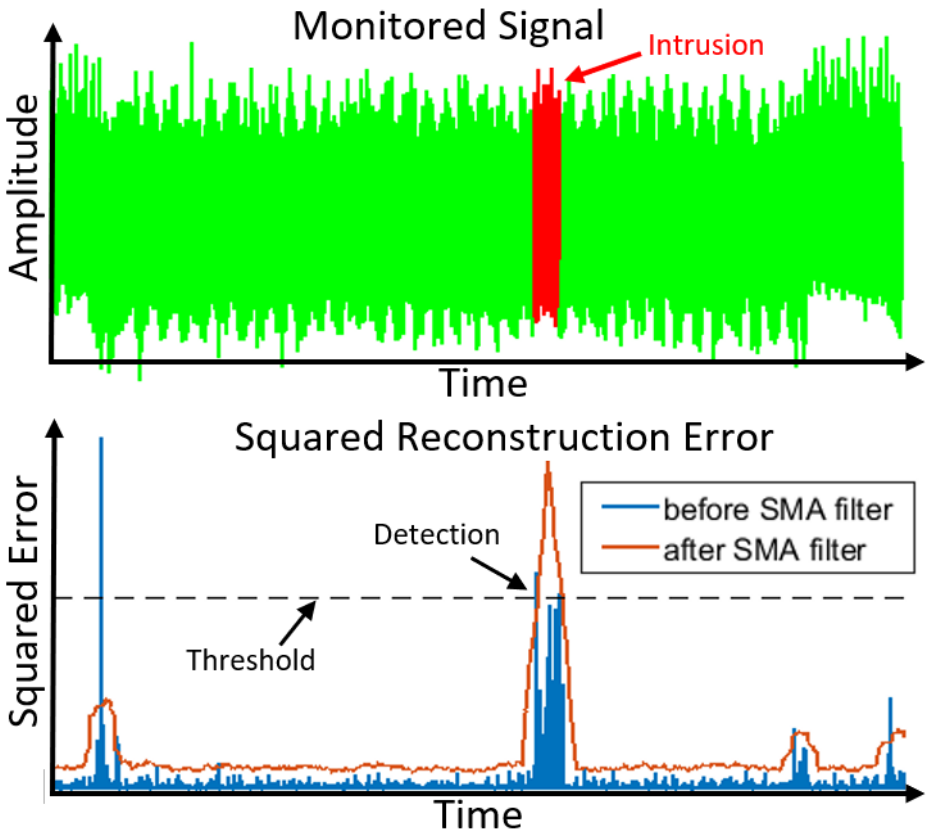


Figure 3.3: Intrusion detection from reconstruction error: the squared reconstruction error is passed through an SMA filter to reduce false positives.

Detection

The detection continuously compares the monitored signal with the reconstructed signal. Specifically, we compute the per-sample reconstruction error as the squared difference between samples of the monitored and the reconstructed signal:

$$e(n) = (x(n) - y(n))^2 \tag{3.4}$$

Here, $x(n)$, $y(n)$, and $e(n)$ denote the monitored signal, the reconstructed signal, the squared reconstruction error signal, respectively and n denotes the sample-index.

The detection algorithm is illustrated in Figure 3.3. Due to considera-

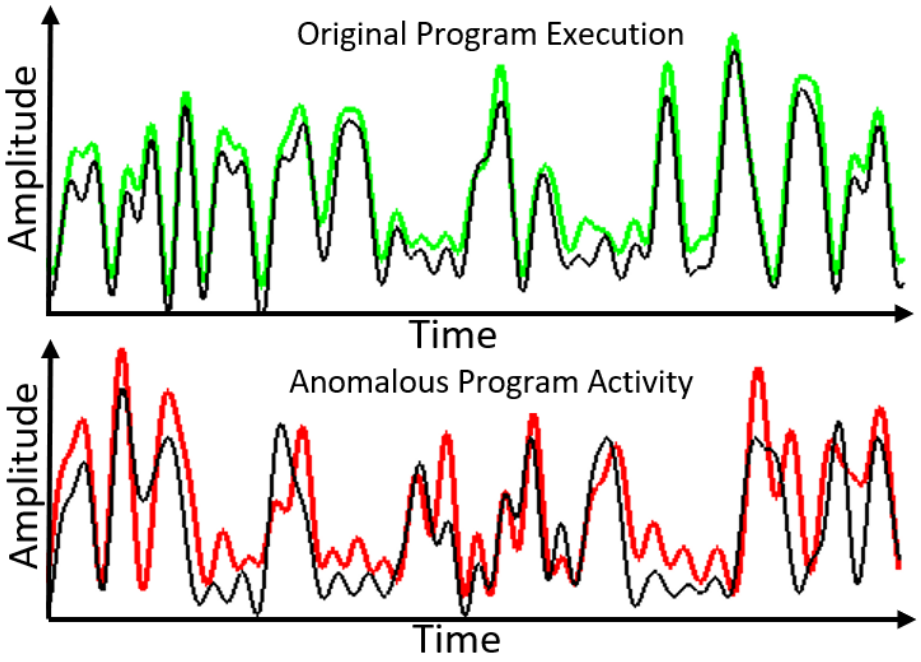


Figure 3.4: Top green curve: original program execution; Bottom red curve: program execution with intrusion; Black curve: IDEA-reconstructed signal. Note that black curve better matches with green than with red curve.

tions presented in Section 3.3.4 below, we then apply an L-samples long Simple Moving Average (SMA) filter to the signal $e(n)$, yielding the filtered signal $\tilde{e}(n)$:

$$\tilde{e}(n) = \frac{1}{L} \sum_{i=0}^{L-1} e(n-i) \tag{3.5}$$

Finally, we set a threshold on $\tilde{e}$, and whenever this threshold is breached, we report an intrusion. Figure 3.4 illustrates how reconstructed curves based on IDEA algorithm match the original program execution vs. the execution with malware. From the plot we can observe that the reconstructed signal deviates significantly from the execution with malware compared to the deviation from the normal execution, hence allowing us to detect the malware.

3.3.4 System Parameters

The performance of IDEA depends on a number of system parameters, such as word-length l, word-shift s, and order of the SMA filter L. The rest of this section is a discussion of how these system parameters are chosen.

Word-Length

The word-length l has an impact on the performance of the proposed system. The optimal word-length l is a tradeoff between confidence in a match (the longer the word, the more reliable the match) and likelihood of a good match (the shorter the word, the more likely it is to find a dictionary word that matches it well).

Therefore, the word-length l should be long enough to avoid good matches among a set of unrelated signals. To achieve that, a word in a dictionary should represent a relatively long sequence of processor instructions or hardware activity. This ensures that it is unlikely that a non-trained program will produce a sequence of executed processor instructions that is an excellent match for any dictionary entry of a trained program.

On the other hand, the word-length l should be short enough so that random events, such as cache misses or interrupts, do not preclude good matches. For example, if we use a word that is very long, it will be difficult to find good matches in the dictionary of any reasonable size. This is because different inputs and hardware activities result in different signals when executing the same code, a reasonable-sized dictionary can contain only a small subset of the possible valid words, and a (long) window of the monitored signal will likely exhibit many input-dependent and hardware behavior that do not match any of the dictionary words. By using a smaller word length we limit the number of word variants that can be

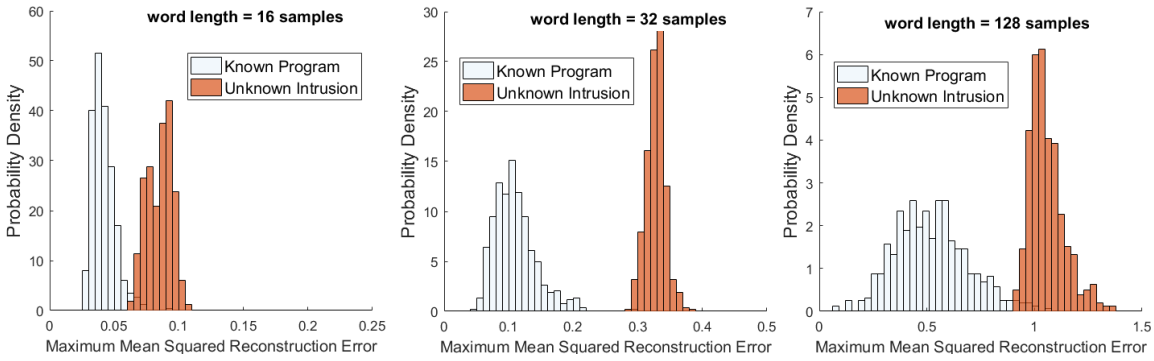


Figure 3.5: Histogram for maximum mean squared reconstruction error with different word-lengths.

produced, which reduces the dictionary size required for "full" coverage of these variants. Even when dictionary coverage of word variants is not complete and a window of the monitored signal has a set of input-dependent and hardware behavior that is not represented in the dictionary, a smaller word length increases the probability that the dictionary contains a word that matches the window for most of its duration, and thus still produces a reasonably small reconstruction distance.

To estimate the optimal word-length l, we insert snippets of untrained signals into the trained or trusted reference signals. First, we record EM signals by executing a benchmark program with different inputs. Next, we follow a 10 fold cross-validation to test each of these signals with and without an "untrained" insertion from a different benchmark program. Here, signals without insertion represent class 0 or "known", while signals with insertion correspond to class 1 or "intrusion". Figure 3.5 shows the histograms corresponding to "known" and "intrusion" with different word-lengths. For $w = 16$ samples, the Maximum Mean Squared Reconstruction Error (MMSRE) is low for both known (or trained) and intrusion (or untrained) signals (i.e., even an untrained signal can be matched with words in the dictionary). As a result, the two histograms overlap. However, for

$w = 32$ samples, MMSRE corresponding to the known signal is significantly lower than that of the intrusion, and there exists a clear threshold between the two classes. When the word-length is much larger, i.e., $w = 128$ samples, MMSRE for both known and intrusion signals gets much higher (i.e., even a trained signal cannot be matched with low Euclidean distance). Hence, the two histograms cannot be separated anymore.

These experimental evaluations reveal that for any intrusion larger than 256 samples, IDEA can achieve Area Under the Curve (AUC) better than 0.9995 on the Receiver Operating Characteristic (ROC) curve for any word-length between 32 to 64 samples. If not specified differently, the rest of the paper assumes word-length $w = 32$ of samples. This corresponds to 5 μs of execution time, which is about 250 processor clock cycles on the FPGA board.

Word-Shift

Another parameter that impacts the performance of IDEA is word-shift. Each “word” in the dictionary has to be shifted some number of samples from the previous one in order to compensate for hardware activities such as cache hit or miss. We estimate the optimal word-shift s through experimental evaluation. Again, we exploit a 10 fold cross-validation in which snippets of insertions from an “untrained" program are treated as intrusions. Figure 3.6 shows the ROC curve for different word-shifts for an intrusion of 128 samples. It is clear that $s = 1$ performs the best, and the larger s results in smaller AUC. This is intuitive as $s = 1$ mimics shift-invariant signal matching most closely. However, it should be noted that the detection performance for $s = 2$ is comparable to that of $s = 1$. Hence, $s = 2$ can be exploited to reduce the computational requirements. The

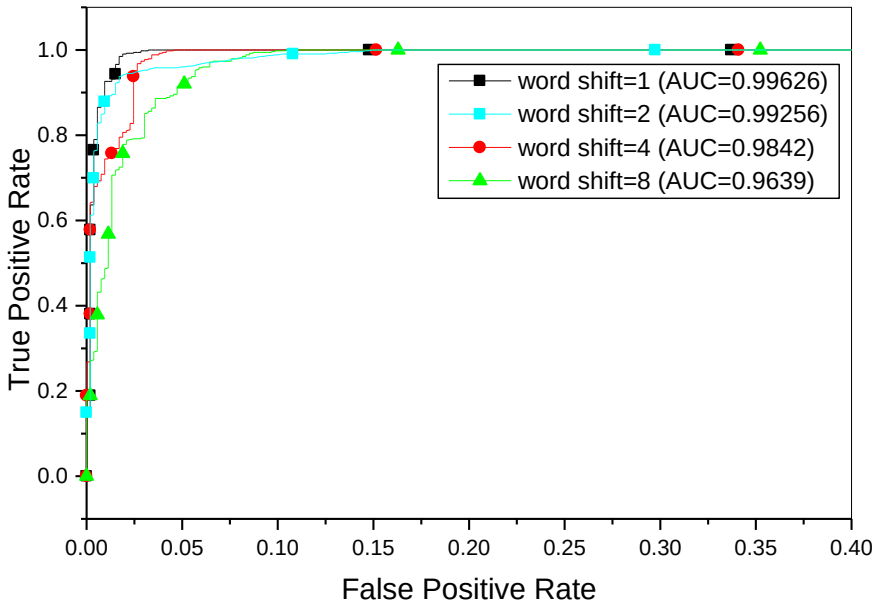


Figure 3.6: ROC curves for intrusion detection with different word-shift.

number of entries in the dictionary would be roughly halved for $s = 2$ compared to $s = 1$. So, the memory requirement would be halved, and consequently, so would be the computational time. Nevertheless, as we intend to highlight the performance of our system, we use $s = 1$ in the remainder of this paper.

Filter Order

In order to justify the use of the SMA filter and to determine its optimal length, consider the following detection problem. Let $\epsilon(n) \triangleq x(n) - y(n)$ denote the error signal, defined as the difference between the monitored and reconstructed signals. Observing an L-samples segment thereof $\epsilon(n - L + 1), \ldots, \epsilon(n)$, we wish to decide whether:

- H_0 : This is a valid program execution segment; or
- H_1 : This is an intrusion code segment.

We begin by attributing two simplified statistical models to the error signal

under each hypothesis: $\epsilon(n)$ is assumed to be an independent, identically distributed (iid) zero-mean Gaussian process, but with a different variance under each of the different hypotheses:

- $H_0 : \epsilon(n) \sim \mathcal{N}(0, \sigma_0^2)$
- $H_1 : \epsilon(n) \sim \mathcal{N}(0, \sigma_1^2)$

where $\sigma_0^2 < \sigma_1^2$ are fixed variances (presumed known, for now). The Likelihood Ratio Test (LRT) for deciding between the two hypotheses then takes the form:

$$\frac{f(\epsilon(n-L+1), \ldots, \epsilon(n)|H_1)}{f(\epsilon(n-L+1), \ldots, \epsilon(n)|H_0)} \underset{H_0}{\overset{H_1}{\gtrless}} \eta, \tag{3.6}$$

where $f(\cdot|H_i)$ denotes the conditional joint probability distribution function (pdf) of the observations given $H_i, i = 0, 1$, and η is a threshold value. Substituting Gaussian distributions and taking the log we get

$$-\frac{L}{2}\log(\sigma_1^2) - \frac{1}{2\sigma_1^2}\sum_{i=0}^{L-1}\epsilon^2(n-i) + \frac{L}{2}\log(\sigma_0^2) + \frac{1}{2\sigma_0^2}\sum_{i=0}^{L-1}\epsilon^2(n-i) \underset{H_0}{\overset{H_1}{\gtrless}} \log(\eta), \tag{3.7}$$

which can be further rearranged as

$$\frac{1}{L}\sum_{i=0}^{L-1}\epsilon^2(n-i) \underset{H_0}{\overset{H_1}{\gtrless}} \tilde{\eta} \triangleq \frac{\kappa\sigma_0^2}{\kappa - 1}\left(\log(\kappa) + \frac{2}{L}\log(\eta)\right), \tag{3.8}$$

where $\kappa \triangleq \sigma_1^2/\sigma_0^2 > 1$ denotes the ratio between the two variances. This means that the average of squared samples of $\epsilon(n)$ over the L-samples observation interval is to be compared to some threshold $\tilde{\eta}$. This average is precisely $\tilde{e}(n)$, the output of a length-L SMA filter in (4.10), with $e(n) \triangleq \epsilon^2(n)$.

To determine an optimal value for L, note first that the mean and variance of $\tilde{e}(n)$ under H_i are (resp.) σ_i^2 and $\frac{2}{L}\sigma_i^4$, $i = 0, 1$. We note further, that if L is sufficiently large (say, larger than 10 or so), considering the

Central Limit Theorem, the distribution of $\tilde{e}(n)$ under each hypothesis is approximately Gaussian. Consequently, the false positive and false negative probabilities can be shown to decay monotonically in L. Therefore, to have them minimized, L should take the largest possible value that does not breach the H_1 model. Namely, L has to be chosen as the full length of the shortest possible intrusion. Since that length is not known a priori, we chose to set the filter order equal to the shortest length of insertion that we intend to detect. If not specified differently, we use $L = 256$ as the filter length.

Note that in reality, the situation is somewhat more complicated than described above. First, the iid Gaussian signal model for $\epsilon(n)$ is inaccurate, as its samples are expected to be correlated and not necessarily Gaussian distributed; The variances σ_1^2 and σ_0^2 would rarely be known; And reconstruction errors may be very high for a few brief, sporadic segments, even under H_0, resulting either from lack of full coverage in the training phase (i.e., lack of appropriate training examples that follow the same control flow path as the monitored signal) or from the variability of hardware activities between the training signal and the monitored signal. These complications certainly undermine any claim of optimality of the LRT in this case. However, the rationale behind the resulting test remains valid, justifying the use of the SMA filter and the choice of L. For example, the possibility of short occurrences of large errors under H_0 would merely increase the mean and variance of $\tilde{e}(n)$ under H_0, thereby increasing the false positive rate. Nevertheless, the dependence of this rate on L remains monotonically decreasing, still supporting our choice of the largest possible L that does not breach H_1.

3.4 Experimental Evaluations

In this section, we present our experimental results on detecting several different types of malware on different applications and embedded systems. It is important to emphasize that IDEA is not limited to these applications and/or malware, but fundamentally can be applied to any system that has observable EM emanations.

3.4.1 Experiments with Different Malware Behavior

To show the effectiveness of IDEA on detecting different malware behavior, we selected 3 applications from the SIR repository [93]: (*Replace*, *Print Tokens*, and *Schedule*) and implemented three common types of embedded system malware payloads (DDoS attacks, Ransomware attacks, and code modification) on an Altera DE-1 prototype board (Cyclone II FPGA with a NIOS II soft-processor) while executing any of these SIR applications. We selected applications from SIR repository because they are relatively compact (allowing manual checks of our results to get a deeper understanding of what affects the IDEA accuracy), are commonly used to evaluate the performance of techniques that analyze program execution, and have many program inputs available (each taking a different overall path through the program code), so we can use disjoint sets of inputs for training and monitoring and yet have a large number of inputs for training (to improve code coverage) and monitoring (to obtain representative results).

For *DDoS* cyber-attack, we assume that the attacker exploits a vulnerability (e.g., buffer-overflow) to divert the control-flow (e.g., using code-reuse attack) to a code that sends DDoS packets in rapid succession, without waiting for a reply from the target. After sending a burst of packets, the

malware returns execution for a while to the original application, so the device continues to perform its primary functionality. In the case of an embedded device, e.g., Altera DE-1 (Cyclone II FPGA) board, there is no traditional network (e.g., Ethernet) port. So we instead implement the packet-sending activity using the JTAG port.

For *Ransomware* attack, we implement a Cryptoviral [94], that encrypts the victim's data and demands a ransom in return for the decryption key. We assume that the attacker inserted the malicious code through firmware modification. We used *Advanced Encryption Standard*, AES-128 for encryption. Encryption of large files is time-consuming and can be detected easily by IDEA. To make things more challenging, the *Ransomware* we implement encrypts only one encryption block of AES-128, which corresponds to a 16-byte data. Note that more secure ciphers, e.g., AES-256, have larger encryption blocks and are thus easier to detect.

For *code modification*, which is the basis of an important class of malware (APT and firmware modification attacks), we assume that the malware has already successfully modified the source code of the program and that the goal of IDEA is to detect when this modified code executes. For example, in the *Replace* benchmark, there is a function called *subline()*, which is used to search for words in an input string. In a scenario where authorities use this code to look for names in intercepted communication, the attacker's modification of this code would prevent any word that begins with specific initial letters from being reported.

Setup: To observe the EM emanations, we used a 2.4-2.5 GHz 18dBi panel antenna that was placed 1 m, 2 m, and 3 m away from the device (see Figure 3.7). We used a (relatively expensive) Agilent MXA N9020A spectrum analyzer to demodulate and record the EM emanations, primarily so we

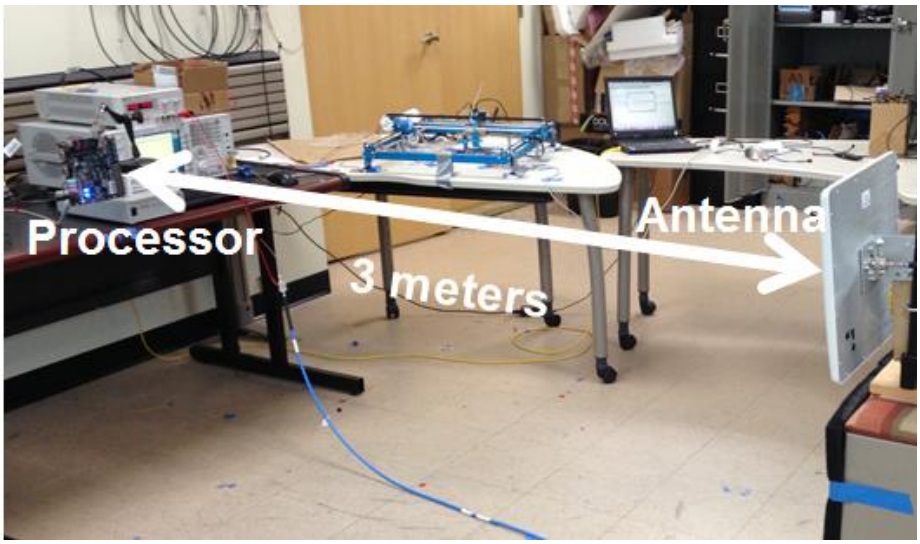


Figure 3.7: Experimental setup used to collect EM traces from distance.

can have more control and flexibility in our investigations. However, in the next section (3.4.2), we will show that a sub-$1000 SDR receiver (USRP-B200mini [95]) can be used instead. To assess IDEA's performance, we have used program input sets from [93]. Malware injections occur in roughly 30% of the runs (randomly selected), and in each injection, the injection time (relative to the beginning time of the run) is drawn randomly from a set of 10 predefined values.

Detection Performance: To evaluate the performance of the intrusion detection system, we apply a 10 fold cross-validation. We execute each benchmark program with different inputs. Some of these executions are infected with malicious intrusions. As the system does not assume any a priori knowledge about the threat models or the intrusions, we do not use any of these infected executions for training the system. Likewise, we select all the system parameters through experiments with "untrained" insertions from a different benchmark program, without using any actual infected signal. The non-infected signals are randomly divided into 10 roughly equal-sized subsets. In each fold, we test one of these subsets

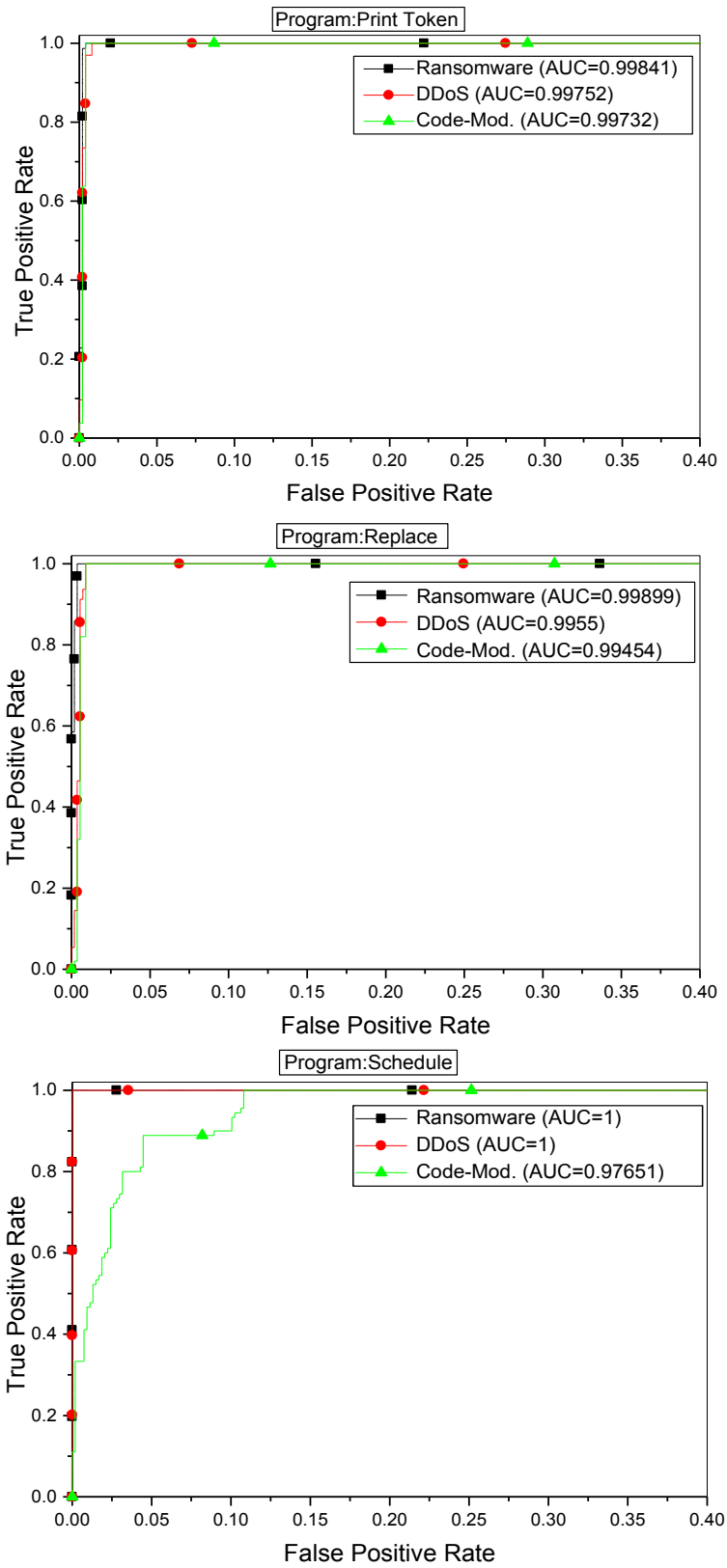


Figure 3.8: Receiver Operating Characteristic curves for intrusion detection on three different benchmark programs for different variants of malware.

along with the infected signals, while the rest of the non-infected signals are used for training the system. The objective of the experiment is to evaluate how successfully the system can identify and differentiate between the infected and the non-infected signals.

We follow this procedure with three different benchmark programs, namely *Print Tokens*, *Replace*, and *Schedule*. For *Print Tokens*, a total of 637 executions were recorded, out of which 142 were infected with different variants of malware - 68 Ransomware, 66 DDoS, and 8 code modification. In case of *Replace*, we recorded 691 executions including 138 infected ones out of which 68 were Ransomware, 65 were DDoS, and 5 were code modification. For *Schedule*, we collected 681 executions where 144 of them were infected with 68 Ransomware, 67 DDoS, and 9 code modification.

Figure 3.8 plots the ROC curves for IDEA intrusion detection on different benchmark programs with different variants of intrusions. It shows excellent performance in all three benchmark programs, with Area Under the Curve (AUC) very close to 1 for Ransomware and DDoS malware. Code modification creates a much smaller change in the program's execution and is much harder to detect. However, IDEA still detects it with an AUC > 97.5%. Specifically, detection of all code modifications is achieved by tolerating a false positive rate of no more than 1% in *Print Tokens* and *Replace*, and no more than 12% in *Schedule*.

Detection at Different Distances: Next, we test IDEA at three different distances (1 m, 2 m, and 3 m) from the monitored device. The results (Figure 3.9) show that the performance of IDEA remains stable over different distances. In fact, for all three benchmark programs, the performance is quite similar at 1 m and 2 m, with AUC better than 99.5%. The perfor-

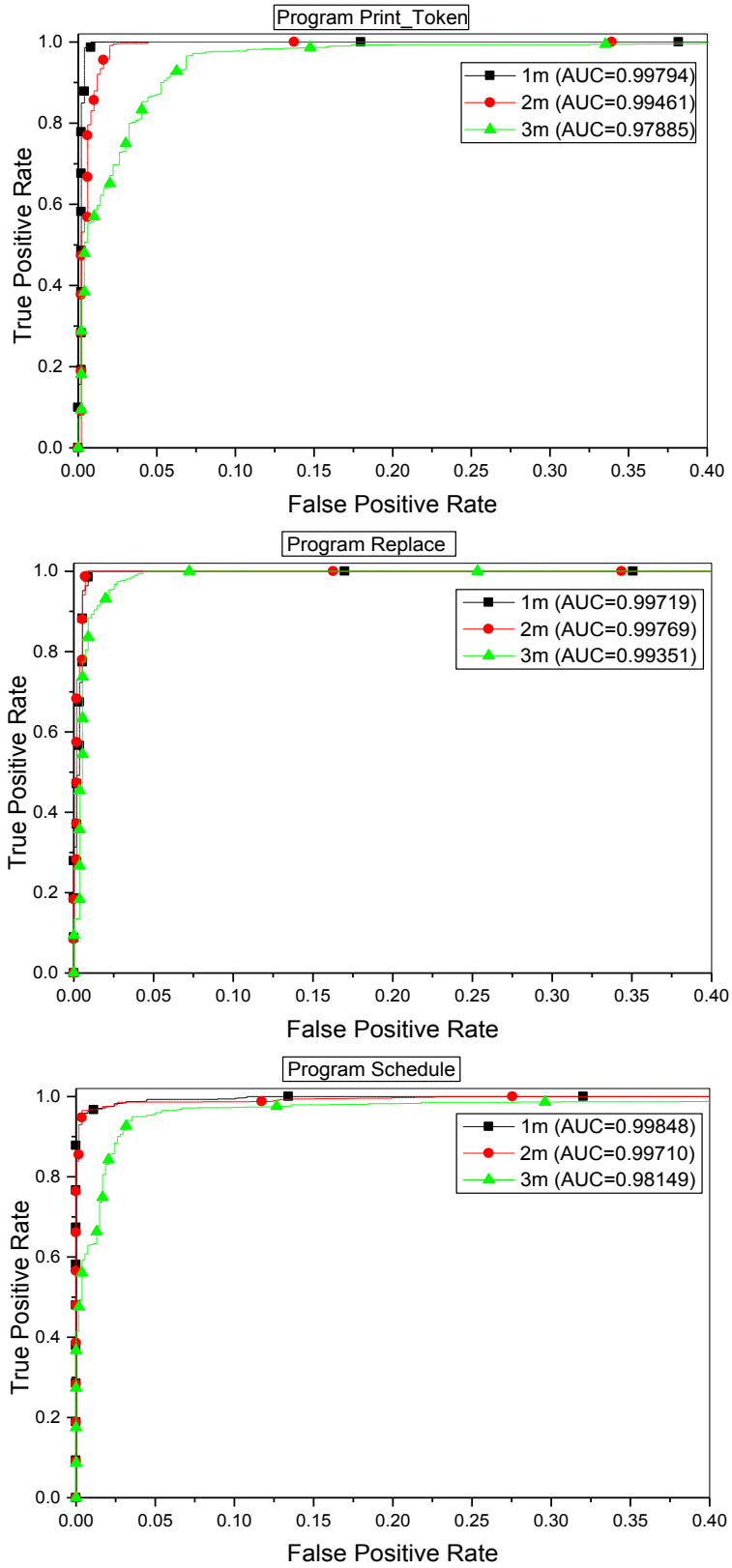


Figure 3.9: Receiver Operating Characteristic curves for intrusion detection on three different benchmark programs from different distances.

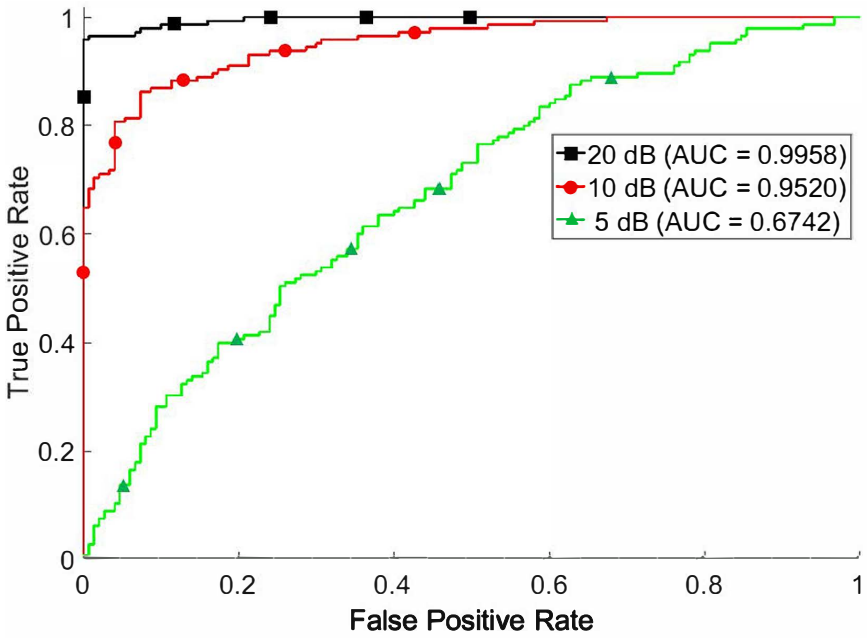


Figure 3.10: Receiver Operating Characteristic curves for intrusion detection at different SNR.

mance degrades somewhat at 3 m, achieving 99% AUC for *Replace* and 98% AUC for *Print Tokens* and *Schedule*. The degradation in accuracy at the 3 m distance is mainly due to a reduced signal-to-noise ratio (SNR) as the monitored signal weakens with distance. We believe that these results can be improved by using customized (higher-gain) antennas and low-noise amplifiers.

Detection at Different SNR: To evaluate IDEA in presence of environmental EM noise, we apply Additive White Gaussian Noise (AWGN) to the monitored signal, and test IDEA by monitoring the benchmark application *Replace* at three different SNR (20 dB, 10 dB, and 5 dB). The results are shown in Figure 3.10. Experimental evaluation demonstrates that IDEA is robust against EM noise. In fact, IDEA achieves an excellent performance (AUC > 99.5%) at 20 dB SNR. Furthermore, at 10 dB SNR, IDEA can still detect intrusions with an AUC > 95%. However, at 5 dB SNR, the detection performance degrades to a 67.4% AUC.

In addition, IDEA is inherently robust against out-of-band EM interference from adjacent devices. IDEA monitors the target device at a frequency band around its processor's clock frequency. Any EM interference with frequencies outside this monitored band is blocked by the anti-aliasing filter during the analog-to-digital conversion (ADC). Since each device emits EM signal at its own clock frequency, interference is limited and can be filtered out using signal processing.

3.4.2 Experiments with Cyber-Physical Systems

To emphasize the practicality of IDEA and to demonstrate its ability to successfully detect real malware on real CPSs, we use IDEA to monitor three industrial CPSs implemented on a well-known embedded system, Arduino UNO. We use a control-flow hijacking attack, an advanced persistent threat (APT), and a firmware modification attack on these CPSs for evaluation.

The first CPS we use is called a *SyringePump*. A SyringePump dispenses or withdraws a precise amount of fluid, e.g., in hospitals for applying medication at frequent intervals [96], and is a representative of a medical CPS. The device typically consists of a syringe filled with medicine, an actuator (e.g., stepper motor), and a control unit (Arduino UNO) that takes commands and produces controls for the stepper motor (a sample of this CPS can be found in [96]). We implement a *control-flow hijack* attack on this system by exploiting an existing buffer-overflow vulnerability in a subroutine (*serialRead()*) that reads the inputs which causes the program's control-flow to jump to an *injected* malicious code. We assume that the adversary is interested in disrupting the correct performance of the system by dispensing/withdrawing an unwanted amount of fluid, which could

cause significant damage to the patient. Thus, the injected code causes the syringe to dispense a random amount of fluid. The buffer-overflow is implemented by sending a large input to overwrite the stack, followed by the address of the "injected" malicious code that overwrites the actual return address of the *serialRead()*.

The second system is a proportional-integral-derivative (PID) controller that is used for controlling the temperature of a soldering iron. This type of system could also be used to control the temperature or any other critical value in other settings, such as a building or an industrial process, and thus is representative of a large class of industrial CPSs. Using a feedback loop and a history of previous temperatures, the system keeps and/or changes the temperature to a desired temperature (a sample of this CPS can be found in [97]). To implement an APT on this application, we assume that the adversary's malware (like in Stuxnet) has already infiltrated the system and can reprogram the device. The adversary's goal is to change a critical value under some conditions, which in turn can cause damage to the overall physical system. In our evaluation, we made a malicious modification to the source code so that the temperature history is altered under a specific condition (e.g., for a specific model number). Consequently, the system will set a wrong temperature. The injected code is only 2 lines of code (i.e., IF(X) THEN LASTTEMPHISTORY = RANDOMVALUE).

The final system in our evaluation is a robotic arm. Robotic arms are often used for manufacturing and are critical components of many modern factories. Robotic arms typically receive inputs/commands from a user and/or sensors and move objects based on these inputs. There is a growing concern in the security of these CPSs since they are typically connected to the network and are exposed to cyber-threats (e.g., [98]). A

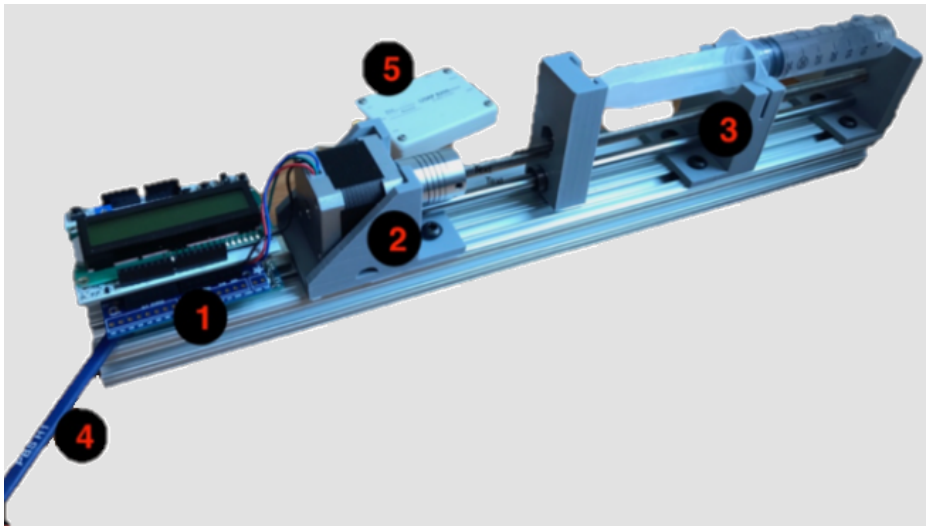


Figure 3.11: Experimental setup for the SyringePump: (1) Arduino device with LCD, (2) stepper motor, (3) syringe, (4) magnetic probe and (5) software-defined radio.

simple implementation of such a robot can be found in [99]. For this system, we implement a *firmware modification* attack, where we assume that the reference libraries (e.g., library for *Servo*, *Serial*, etc.) are compromised (this can be also considered as a *zero-day* vulnerability). Note that we assume that IDEA's training contains the "unmodified" version of these libraries (baseline reference data). In this attack, we modify a subroutine (*writeMicroseconds()*) in Arduino's Servo library [100] by adding an extra *if* condition to change the speed of the Servo motor randomly and reprogram the system with this compromised library, assuming that the adversary is interested in causing a malfunction in arm's movement.

Setup. An Arduino UNO with an ATMEGA328p microprocessor clocked at 16 MHz is used to implement the CPSs. A magnetic probe is used to receive EM signals from the device. Figure 3.11 shows the experimental setup for the *SyringePump*. For all measurements, we use a commercially available SDR receiver (Ettus Research B200-mini) to record the signal.

Table 3.1: Experimental results for three malware on three CPSs.

System	AUC	Malware Type
SyringePump	> 0.999	Control-flow hijack
PID Controller	> 0.999	APT
Robotic Arm	> 0.999	Firmware Modification

B200-mini costs significantly lower than a spectrum analyzer and makes IDEA a practical option for monitoring security-critical systems. For each CPS, we use 25 randomly selected signals for training and 25 malware-free and 25 malware-afflicted signals for testing.

Detection Performance. Table 3.1 summarizes the detection accuracy of IDEA on the three CPSs. As seen in the table, in all cases, IDEA has successfully detected every instance of malware without reporting any false positive. This makes IDEA a promising framework for monitoring critical CPSs, where a high detection accuracy is required while having a low false-positive rate. Note that in all cases, the runtime for the malicious code is significantly less ($< 0.01\%$) than the overall runtime of the application.

3.4.3 Experiments with IoT Devices

To demonstrate the robustness of IDEA, we also use it to monitor an A13-OLinuXino (Cortex A8 processor) IoT board. Unlike the FPGA-based system that runs the application "on bare metal," this board runs a Linux operating system (OS). The defensive mechanisms already present in the OS make it harder to inject prototype malware activity. Instead, we model malware injection by injecting snippets of signals from a different (not-trained-on) program. For this experiment, we use *Replace* as the reference program, on to which signal-snippets from *Print Tokens* were inserted as anomalous (not-trained-on) signal. This approach also allows injections of any chosen duration and the use of different signals for different injec-

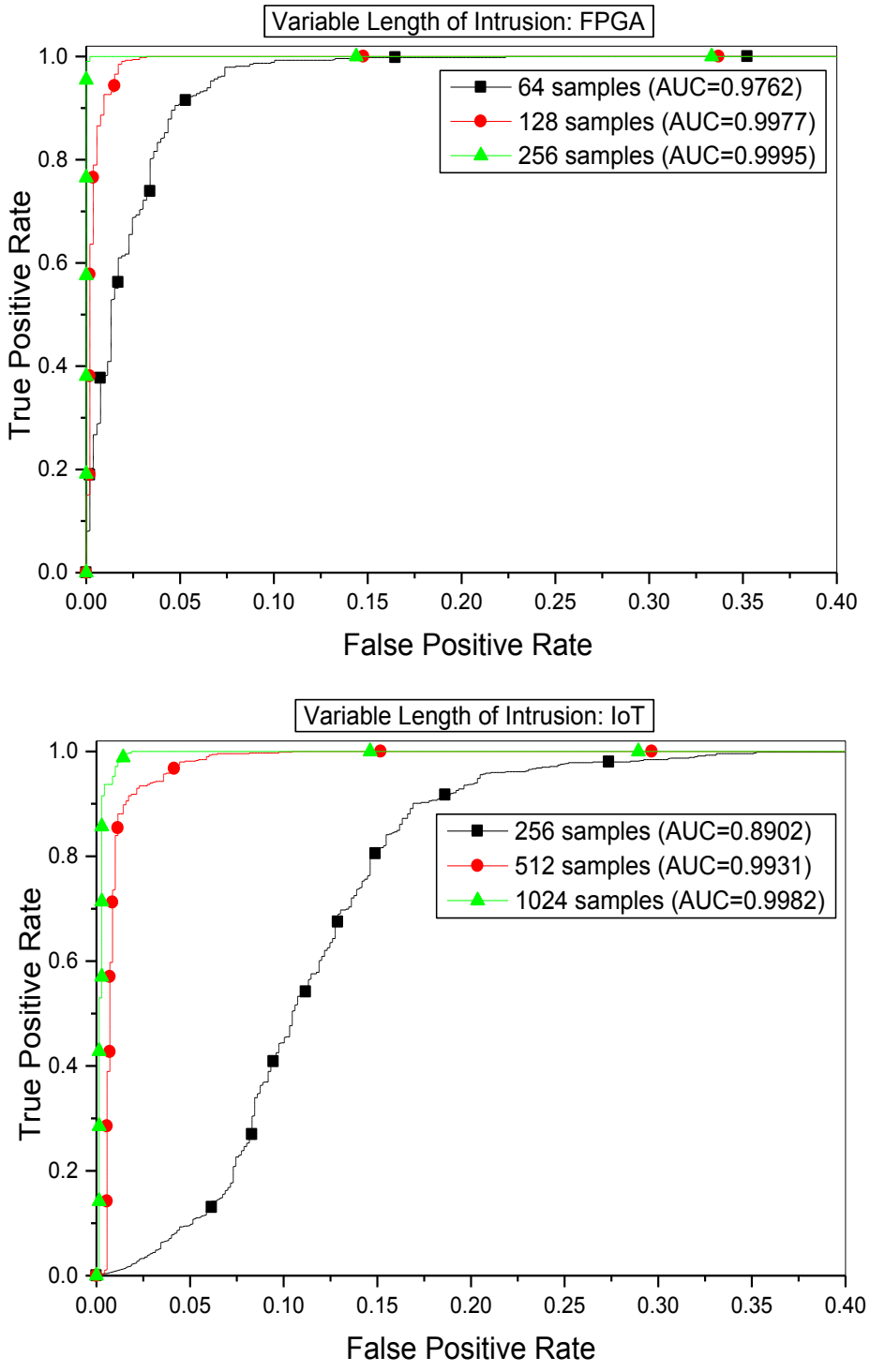


Figure 3.12: Receiver Operating Characteristic curves for intrusion detection on an FPGA (top) and on an IoT Device (bottom).

tion instances. In contrast, construction of even one short-duration actual malware instance is very challenging. For example, a single packet sent in a DDoS attack, or single-block encryption in Ransomware, lasts much longer than any of our signal-snippet injections.

To allow a direct comparison between our real-malware and signal-snippet injections, we also perform signal-snippet injection experiments on the DE-1 FPGA board. We use 10-fold cross-validation, and test signals from a trained benchmark program *Replace*, with or without insertions or intrusions from *Print Tokens*. Figure 3.12 shows the experimental results. We can observe that intrusions longer than 256 samples (i.e., 200 instructions or 40 μs length) on the FPGA are detected with an AUC of 99.95%. For the IoT board, an AUC better than 99.8% is achieved for intrusions with at least 1024 samples (i.e., 800 instructions or 7.94 μs length). The difference in duration of the intrusion that is needed to achieve the same AUC on the two devices is mainly due to OS activity that is present on the IoT board and absent on the FPGA board. This OS activity introduces variation in the signals, increasing reconstruction error even for valid executions. This, in turn, raises the reconstruction error threshold for reporting an anomaly at a given confidence level, so more anomalous samples are needed to reach this increased reconstruction error threshold.

3.5 Summary

In this chapter, we presented *IDEA* - a novel intrusion detection framework that uses electromagnetic (EM) side-channel signals to protect embedded devices from stealthy attacks. IDEA first records EM emanations from an uncompromised device to establish a baseline of reference EM patterns. Then, IDEA continuously monitors the target device's EM em-

anations, comparing the observed EM emanations against the reference dictionary. When the observed EM emanations deviate significantly from the entries in the reference dictionary, IDEA reports this as an anomaly/intrusion. Finally, we demonstrated IDEA's effectiveness by monitoring different embedded devices and detecting different malicious attacks with excellent accuracy.

A major concern for commercial deployment of the IDEA monitoring system is its cost. However, we envision that IDEA will be deployed to monitor critical and high-assurance CPSs, e.g., critical infrastructures, military systems, hospital equipment, etc. In such scenarios, the cost of deployment (e.g., cost of antenna, software-defined radio, and signal processing) is offset by the cost of the monitored system and by the cost and consequences of the security breach. In addition, the deployment of IDEA is relatively simple; IDEA does not make any change to the monitored system and thus creates no regulatory, safety, or disruption concern for the system. Another important concern with IDEA is its scalability. IDEA requires very high training coverage, which is difficult to achieve for larger programs. However, we exploit software engineering techniques that ensure high path coverage for training. Moreover, IDEA stores EM patterns corresponding to normal program activities in a reference dictionary. This dictionary may grow prohibitively large for larger applications. While we use clustering to keep the dictionary size manageable, future work should investigate feature dimensionality reduction techniques (e.g., principal component analysis) to further optimize the dictionary size without sacrificing the detection accuracy.

CHAPTER 4
MALWARE DETECTION USING NEURAL NETWORK MODEL FOR ELECTROMAGNETIC SIDE-CHANNEL SIGNALS

4.1 Overview

In this chapter, we present a novel malware detection system for critical embedded and cyber-physical systems (CPSs). The system exploits electromagnetic (EM) side-channel signals from the device to detect malicious activity. During training, the system models EM emanations from an uncompromised device using a neural network. These EM patterns act as fingerprints for the normal program activity. Next, we continuously monitor the target device's EM emanations. Any deviation in the device's activity causes a variation in the EM fingerprint, which in turn violates the trained model, and is reported as an anomalous activity.

We evaluate the system with different malware behavior (DDoS, Ransomware, and Code Modification) on different applications using an Altera Nios-II soft-processor. Experimental evaluation reveals that the framework can detect DDoS and Ransomware with 100% accuracy (AUC = 1.0), and stealthier code modification (which is roughly a 5 μs long attack) with an AUC $\approx$ 0.99, from distances up to 3 m. In addition, we execute control-flow hijack, DDoS, and Ransomware on different applications using an A13-OLinuXino - a Cortex A8 ARM processor single board computer with Debian Linux OS. Furthermore, we evaluate the practicality and the robustness of our system on a medical CPS, implemented using two different devices (TS-7250 and A13-OLinuXino), while executing a control-flow

hijack attack. Our evaluations show that our framework can detect these attacks with 100% accuracy.

The major contributions of this chapter are:

- A novel framework that exploits neural network to model the device's EM side-channel signal for non-intrusive, external, and contactless malware detection.
- A training method that (1) models EM signals from an uncompromised reference device, (2) does not require any knowledge of the nature of the malware attack (or its EM signature), and (3) does not require access to the application's source code or control-flow graph.
- A non-causal signal modeling that exploits signal masking to achieve low prediction error for known or 'trained-on' signal patterns and high prediction error for unknown or 'not trained-on' signal patterns.
- Empirical evaluations that demonstrate that (1) the system can detect even stealthy (e.g., 5 μs long) malicious attacks with high accuracy and low detection latency, (2) the system is equally effective for different applications and different embedded devices, (3) the system is robust against noise and interference, and (4) the system can detect malicious activities from up to 4 m distance.

The rest of the chapter is organized as follows: Section 4.2 states the envisioned threat model, Section 4.3 details our framework for malware detection, Section 4.4 presents the experimental evaluations, and finally, Section 4.5 provides the summary.

4.2 Threat Model

We propose a remote monitoring system for critical and high assurance embedded and cyber-physical devices (e.g., medical devices) by leveraging the device's EM side-channel signal. The system can detect malicious attacks through anomalous EM emanation pattern detection. The envisioned threat model includes the following assumptions:

1. The malware detection system does not have any prior knowledge of the nature of the attack or its EM signature(s). The monitoring system only exploits the EM signature(s) of the monitored application. In addition, the detection system may not have access to the application's source code or control-flow graph (CFG). However, we assume that the system has a reference model for malware-free EM signature(s), which we learn by monitoring an uncompromised trusted device. We further assume that the reference model is not compromised by adversarial attacks.

2. The attacker has access to the monitored device. Furthermore, the attacker has prior knowledge of the application, and consequently, can exploit any vulnerability to execute malicious attacks on the system. For instance, the attacker may exploit a buffer-flow vulnerability to launch a separate thread or process to execute a cyber-attack (e.g., DDoS). Likewise, the attacker may execute a control-flow hijack by modifying and disrupting the existing application and its original functionality. In addition, the attacker may even reprogram the application by modifying its source code and execute malicious activity (e.g., code modification attack). However, the proposed malware detection system does not assume any knowledge of the nature of the attack and detects malicious activity through the deviation in the device's EM signature(s).

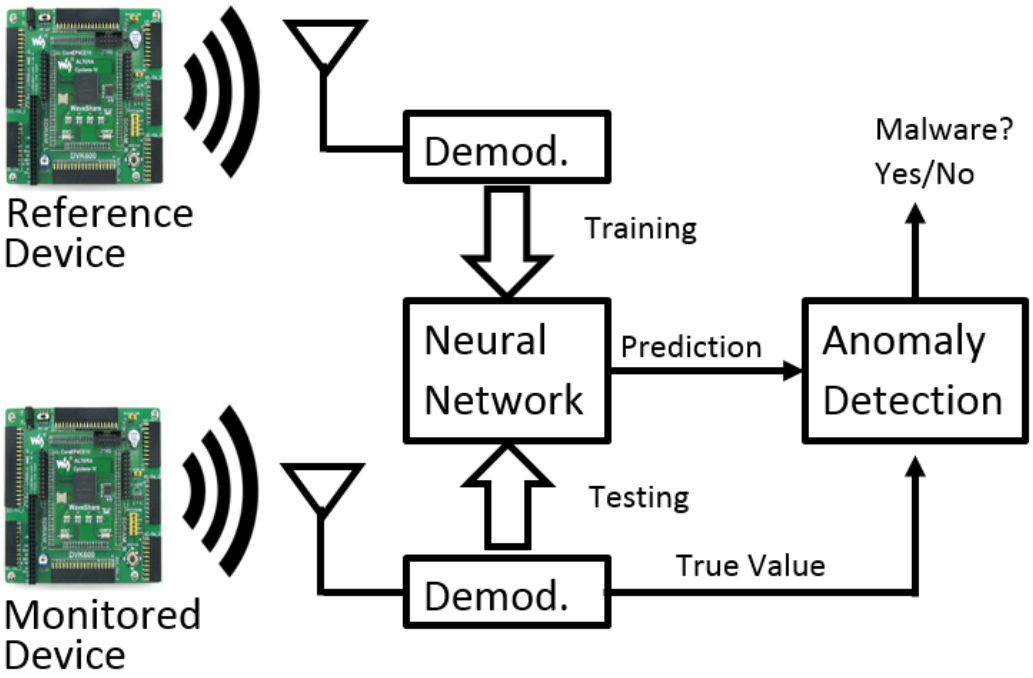


Figure 4.1: Overview of the proposed malware detection system.

4.3 Malware Detection System

We exploit a multilayer neural network for anomalous (hence, potentially malicious) program activity detection through the device's EM side-channel signal analysis. Figure 4.1 demonstrates a high-level overview of the proposed system. During the training phase, the neural network is trained to model the device's EM side-channel signal by executing trusted programs on a reference device. After training, the system is deployed, and it continuously monitors the EM emanation from the target device. When the target device performs any malicious activity, it emanates anomalous (i.e., untrained) EM signal. The deviation in the EM signal causes higher prediction error (as shown in Figure 4.2), and the system reports this as an anomalous program activity. We describe the system in further detail in the following sections.

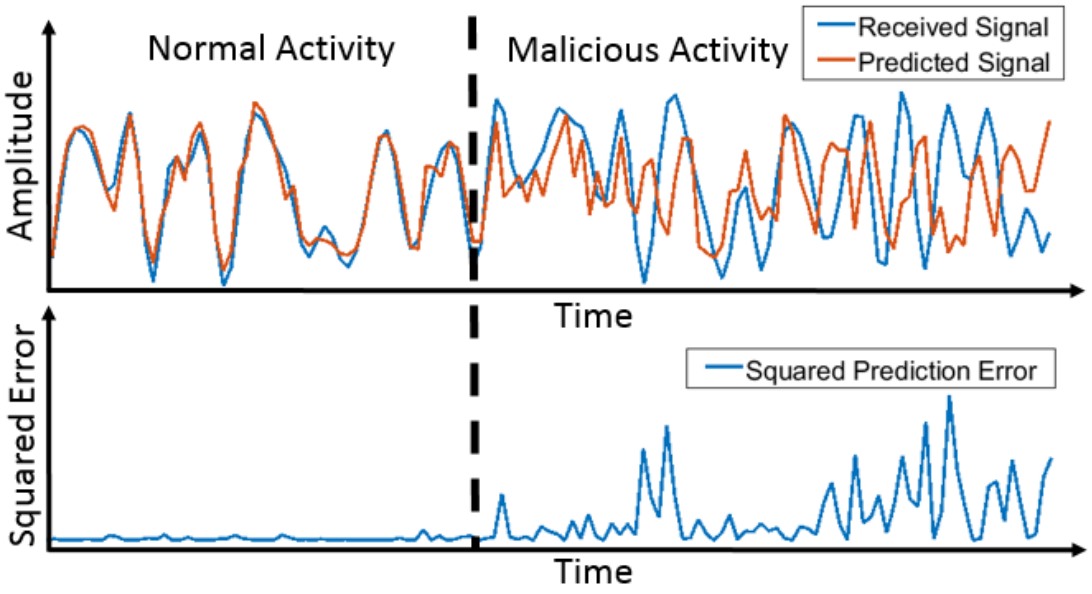


Figure 4.2: Prediction error with normal activity and malicious activity.

4.3.1 Amplitude Demodulation

Before feeding to the neural network, the emanated EM signal is first received through an antenna, amplitude demodulated at the CPU clock frequency, and digitized using an analog-to-digital converter (ADC). At each processor cycle, as the CPU executes new instructions, the states of its internal digital circuits keep changing (i.e., switch on and off). This causes a current at the CPU clock frequency whose amplitude is modulated by the variations of the executed instructions. The carrier modulated current, in turn, causes EM emanation, as it flows within the processor and through the device's printed circuit board (PCB) [91]. Thus to analyze the program-related activities, we demodulate the received signal $r(t)$ at the CPU clock frequency f_c.

$$x_a(t) = |r(t) \times e^{j2\pi f_c t}| \tag{4.1}$$

Here, $x_a(t)$ is the amplitude demodulated analog signal and t denotes the time. The demodulated signal $x_a(t)$ is then passed through an anti-aliasing

filter with bandwidth B and sampled at a sampling period T_s.

$$x_d(n) = x_a(nT_s) \tag{4.2}$$

Here, $x_d(n)$ denotes the sampled signal at sample index n. The anti-aliasing filter cancels unwanted signals with frequencies beyond $f_c \pm B$. Note that, the sampling period T_s is determined by the well known Nyquist criterion $\frac{1}{T_s} > 2B$. Finally, we preprocess $x_d(n)$ by scale normalization.

$$x(n) = \frac{x_d(n)}{max(x_d(n))} \tag{4.3}$$

This ensures that the value of $x(n)$ is between zero and one and also makes the system robust against changes in amplitude of the EM signals (e.g., due to change in the antenna's position, etc.). Finally, $x(n)$ is used as the input for the neural network.

Furthermore, the amplitude demodulation safeguards against minor deviations in the monitored device's clock frequency. The monitored device can have clock frequency shift (due to manufacturing variation) and drift (due to temperature changes). However, the system dynamically detects the device's clock frequency f_c and applies synchronous amplitude demodulation at the detected clock frequency (Equation: 4.1). Consequently, the system is robust against clock frequency shift and drift of the monitored device.

4.3.2 Proposed Neural Network

We use a Multilayer Perceptron (MLP) to model the device's EM side-channel signal. An MLP is a class of feedforward artificial neural network which consists of at least three layers of nodes: an input layer, a hidden layer,

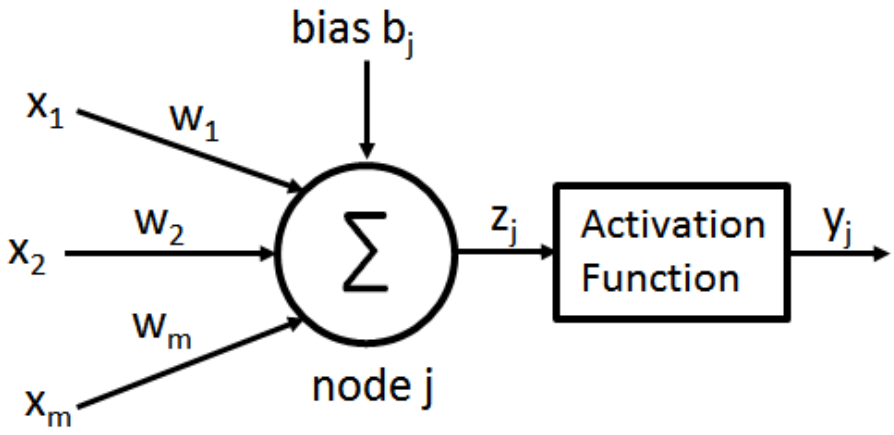


Figure 4.3: Computation performed by a single node.

and an output layer. The output of a node in one layer is typically connected as the input for all nodes in the next layer (i.e., fully-connected layer). As such, it forms a weighted and directed graph and can be exploited to infer complex functions from observations [101, 102].

Each node j computes a weighted sum of its inputs $\mathbf{x}$ and adds a bias b_j to it (as illustrated in Figure 4.3).

$$z_j = \langle \mathbf{w_j}, \mathbf{x} \rangle + b_j \tag{4.4}$$

Here, z_j is the weighted sum of the inputs and the bias at node j, and $\mathbf{x}$ is the input vector, $\mathbf{x} = [x_1, x_2, x_3, ..., x_m]$ and w_j is the vector of connection weights, $\mathbf{w_j} = [w_1, w_2, w_3, ..., w_m]$ and $\langle \cdot, \cdot \rangle$ denotes the scalar product operation. Next, z_j is passed through an activation function (e.g., sigmoid function, hyperbolic tangent function, linear, rectified liner functions, etc.) [103].

$$y_j = \phi(z_j) \tag{4.5}$$

Here, y_j denotes the output of node j after applying the activation function $\phi(\cdot)$. The activation adds non-linearity to the neural network and helps to

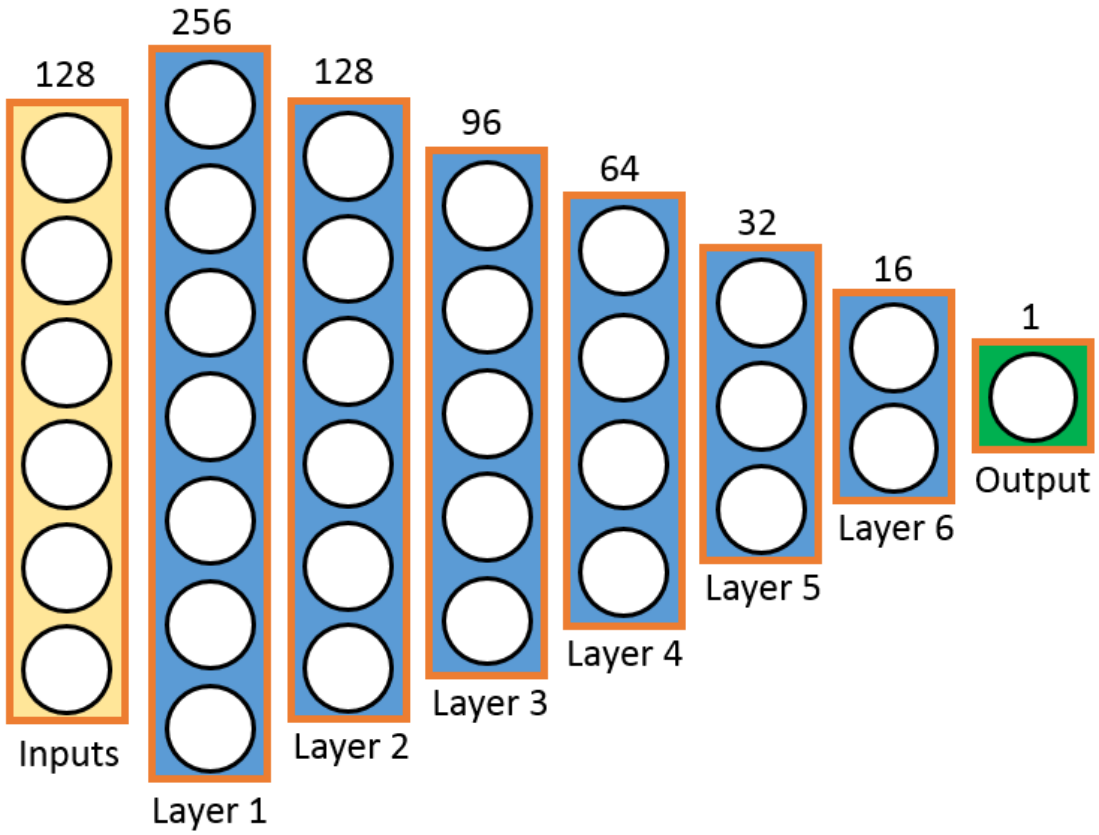


Figure 4.4: Architecture of the proposed multilayer neural network.

model non-linear functions.

While each node performs a simple computation, a neural network can learn to approximate complicated functions by adjusting its weights and biases through training. During training, the network parameters (i.e., weights and biases) are optimized by minimizing a loss function (or cost function) through the backpropagation algorithm [104].

As illustrated in Figure 4.4, the proposed system exploits a neural network architecture that has six fully-connected hidden layers with 256, 128, 96, 64, 32, and 16 nodes, respectively. The input layer has 128 input nodes (i.e., a vector of 128 consecutive samples of $x(n)$), while the output layer has only one output node (i.e., the *estimated* amplitude for sample n). All the hidden layers and the output layer use Rectified Linear Unit (ReLU)

as activation function as rectified linear units have shown to improve performance [105, 106] by mitigating the well-known vanishing gradient [107] problem.

We used MLP to model EM patterns. The other popular network architectures include Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). CNNs are traditionally used for 2D data (e.g., image classification), while RNNs are useful for sequence data (speech recognition, Natural Language Processing, time-series prediction, etc.). However, RNNs are generally harder to train. MLPs, on the other hand, are very flexible and can efficiently learn complex input to output mapping. Thus, we chose MLP due to its simplicity, flexibility, and computational efficiency.

4.3.3 Masking and Prediction

Our proposed neural network models the device's EM side-channel signal and predicts (or outputs) the amplitude (or value) of the EM signal at any instance, given the past and the future EM signal values (or samples) as inputs. The output is

$$y(n) = f(\mathbf{x}^{(n)}) \tag{4.6}$$

where $f(\cdot)$ denotes the neural network model for the device's EM side-channel signal, $y(n)$ is the output (or predicted value), and $\mathbf{x}^{(n)}$ denotes the input vector of the neural network at sample-index n. The input vector $\mathbf{x}^{(n)}$ consists of D samples (i.e., $D = 2(d - k) = 128$ in our system). To better predict $y(n)$, our model uses d *previous* and d *future* samples. However, we hide or mask the k immediate past and the k immediate future samples, as illustrated in Figure 4.5. The main reason for using such a *mask* is that the adjacent samples from an analog time-domain signal, such as an

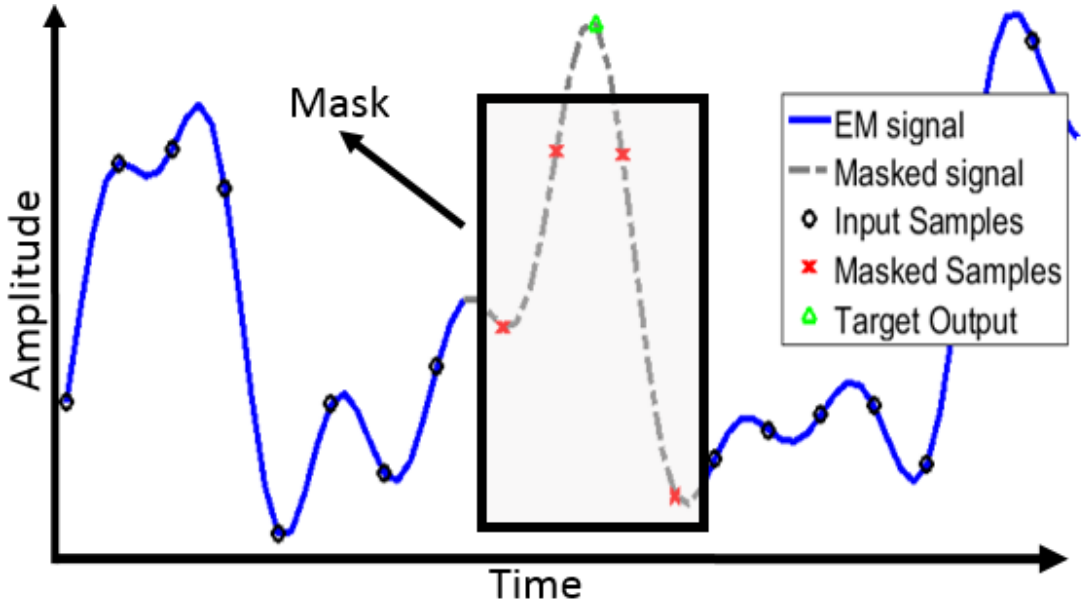


Figure 4.5: Past and future samples are used as inputs (black circles) to predict the target output (green triangle). However, adjacent samples (red crosses) are masked (i.e., not used as inputs).

EM signal, are usually highly correlated, especially at a higher sampling rate. As such, the value of any unknown sample can be predicted through the interpolation of its adjacent samples. However, *interpolation would not be useful for differentiating between normal and anomalous EM signal patterns*. We exploit the neural network model to differentiate anomalous EM signal from normal EM signal through an increase in prediction error. Therefore, we want a prediction model that works well (i.e., low prediction error) for normal (i.e., trained) patterns but results in high prediction error for anomalous (i.e., untrained) patterns. An interpolating function models an unknown sample as a weighted sum of its neighbors. While interpolation could be a good model for predicting highly correlated samples, it would work equally well for both trained and untrained patterns. Thus, the prediction error for the untrained EM signal would be similar to that of the trained signal. Consequently, it would be difficult to differentiate between the normal and the anomalous activity. Therefore, we mask the

adjacent samples to force the neural network to model (or remember) the "normal" EM signal patterns, rather than learning an interpolating function. In our proposed system, we mask 8 immediate past samples and 8 immediate future samples (i.e., $k = 8$), and after removing the immediate 8 samples, use the remaining 64 past and 64 future samples (i.e., $d - k = 64$) as inputs as written in:

$$\mathbf{x}^{(n)} = [x(n-d), x(n-d+1), ..., x(n-k-1), \\ x(n+k+1), ..., x(n+d-1), x(n+d)]. \quad (4.7)$$

It is important to mention that we found that without using masking, the network performs poorly in detecting anomalies.

In the training phase, we collect EM signals by executing malware-free applications on a reference device. We then extract a smaller window from the recorded EM signal. The window consists of $2d+1$ samples, out of which $2(d-k)$ samples are used as the input vector $\mathbf{x}^{(n)}$, 1 sample is used as the target output $x(n)$, while $2k$ samples are masked. Thus, the window acts as a training example (i.e., input and target output pair, $(\mathbf{x}^{(n)}, x(n))$). We then calculate the squared error, $e(n)$, which is computed as the squared difference between the predicted value, $y(n)$, and the true or target output value, $x(n)$, using the given training pair.

$$e(n) = (y(n) - x(n))^2. \quad (4.8)$$

Next, we slide this window through the entire EM signal to get M training examples by setting $n = 1, 2, ..., M$. We use Mean Squared Error (MSE) as

the loss function. MSE is the average of the squared prediction error $e(n)$.

$$\textbf{MSE} = \frac{1}{M}\sum_{n=1}^{M} e(n). \tag{4.9}$$

Here, M is the number of training examples (i.e., the total number of windows), which in our evaluations typically ranges between 2 to 5 million samples. During training, the network parameters are optimized by minimizing the loss function MSE through Stochastic Gradient Descent (SGD) [108] optimization. Note that our Neural Network training is designed such that it minimizes the *average* error, not the error for individual prediction. The main reason is that during training, we observed that individual samples can sometimes experience relatively large errors due to temporary changes in EM signals caused by transient noise (e.g., EMI) and/or micro-architectural events (e.g., cache misses). However, the overall behavior of the signal follows a deterministic pattern for a given application. Thus as the MSE is minimized, the neural network learns to model and predicts the EM signal more accurately (i.e., the prediction error decreases on average).

4.3.4 Anomaly Detection

During the monitoring phase, the trained neural network model is deployed to monitor a target device. The system continuously observes the EM emanation from the device and extracts input and target output pair $(\mathbf{x}^{(n)}, x(n))$ from the EM signal. We use $\mathbf{x}^{(n)}$ as test inputs to predict $y(n)$, and compute the squared prediction error $e(n)$. When the target device performs malicious or anomalous activity (i.e., any activity that the neural network was not trained with), it causes unexpected deviations in the

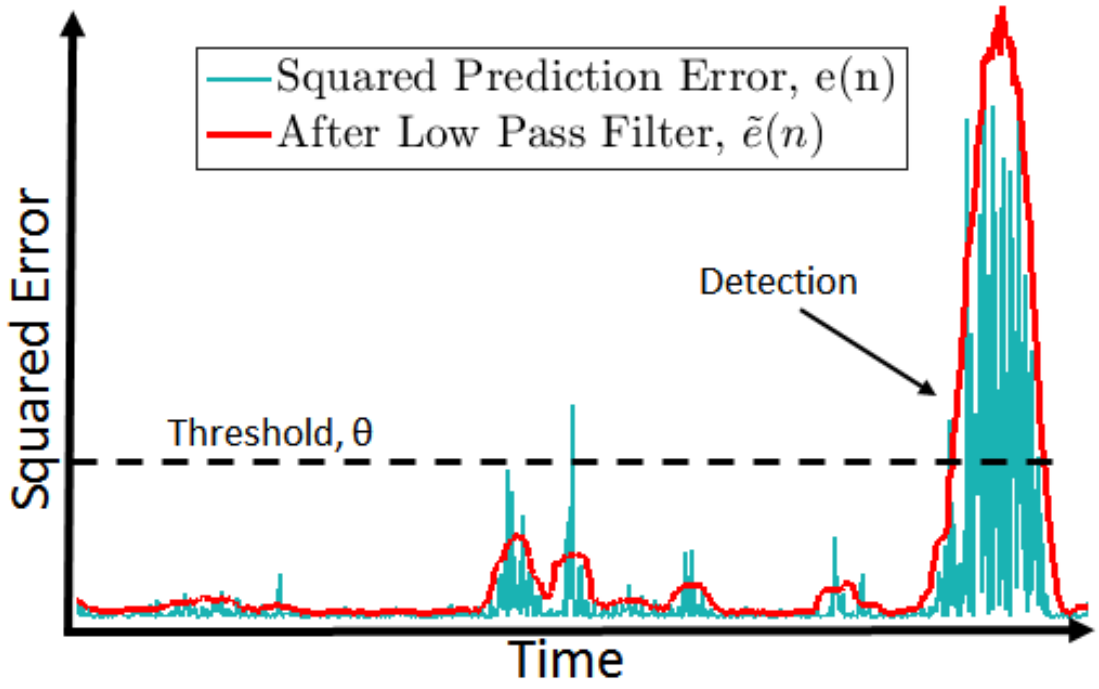


Figure 4.6: Low-pass filtering and thresholding.

device's EM signal. This, consequently, increases the network's squared prediction error $e(n)$. We exploit this fluctuation in $e(n)$ to detect malware execution.

To avoid false positives due to transient noise or variations in hardware activities, which could cause temporary large errors, we low-pass filter the squared prediction error $e(n)$ and apply thresholding to detect anomalous program behavior. Figure 4.6 shows an example of how filtering and thresholding can be helpful to avoid false positives while maintaining the accuracy. We apply an $2N+1$ samples long Moving Average (MA) filter (as a low pass filter) to the signal $e(n)$, yielding the filtered signal $\tilde{e}(n)$:

$$\tilde{e}(n) = \frac{1}{2N+1} \sum_{i=-N}^{N} e(n-i). \tag{4.10}$$

This low pass filtering results into a bi-modal Probability Density Function (PDF) (as shown in Figure 4.7), where the squared prediction error $\tilde{e}(n)$

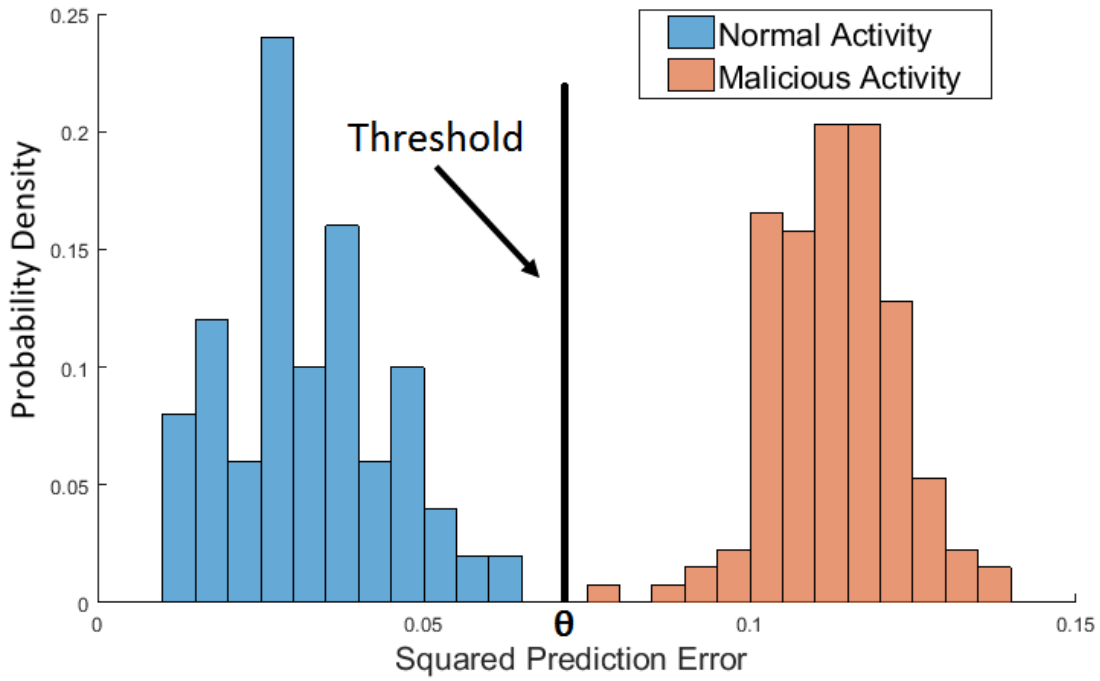


Figure 4.7: Threshold selection using PDF of squared prediction error for normal and malicious program activity.

for normal and malicious program activity can be separated by a threshold θ . Thus, we set a threshold θ on $\tilde{e}(n)$ between the two PDFs, and report anomalous program activity whenever $\tilde{e}(n) > \theta$.

4.3.5 System Parameters

The performance of the detection system depends on a number of system parameters, such as the length of the input vector D, the size of the mask k, the moving average filter parameter N, and the threshold parameter θ. In this section, we discuss how these parameters are chosen and their impacts on the system performance.

Input Vector Length D: The EM signal represented by the input vector provides a "context" for the prediction. More specifically, the Neural Network exploits the past and the future EM patterns to predict the present

EM amplitude. While a larger value for D increases the "context", this also adds to the complexity of the Neural Network and may lead to overfitting. Thus, from empirical evaluation, we use $D = 128$.

Mask Size k: The adjacent samples of the EM signal are more correlated at a higher sampling rate (i.e., with a lower time-gap between two adjacent samples). Thus, intuitively the mask (or k) should be larger with a higher sampling rate. However, a mask that is too large may overshadow the "context", and interfere with the prediction. We monitored FPGA, TS-7250, and A13-OLinuXino with 10 MHz bandwidth (i.e., 5 MHz bandwidth on either side of the clock frequency). Thus, we used the same mask (k = 8) throughout all experiments.

Moving Average Filter Parameter N: The moving average filter helps to reduce false positives due to unpredictable variabilities in hardware activities (e.g., cache misses). These variabilities can cause transient yet high-valued prediction errors. As such, the PDF of the squared error for normal activity resembles an exponential function with a long-tail (as shown in Figure 4.8). This tail overlaps with the PDF of the malicious activity and consequently generates a lot of false positives. However, the MA filter reduces the false positives by transforming the PDF into a symmetric (Gaussian-like) function. With increasing N, the function gets sharper with a shorter tail and results into fewer false positives (i.e., less overlap with the PDF of the malicious activity). However, this reduction of the false positives comes at the cost of increased detection latency. Furthermore, shorter malicious activities (e.g., intrusions that are shorter than N samples) may go undetected. Thus, the optimal N is a trade-off between

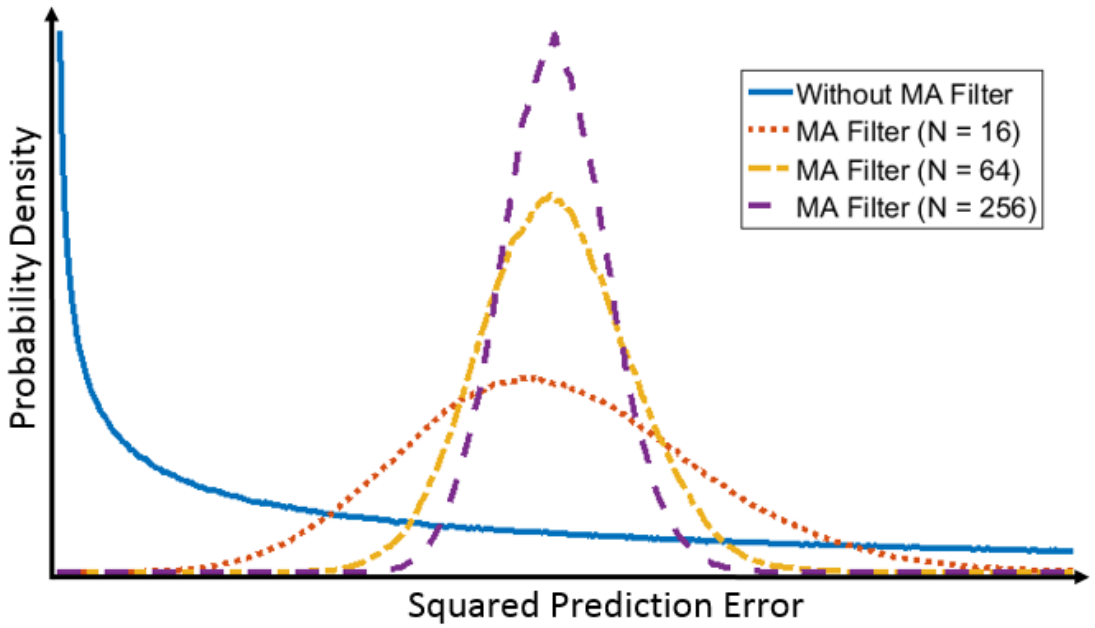


Figure 4.8: Probability Density Function of the squared prediction error for normal program activity with and without moving average filter.

reliable detection (low false positives) and detection latency. In our experiments, we used $N = 64$ for monitoring FPGA, and $N = 1024$ for monitoring TS-7250 and A13-OLinuXino board. The higher order MA filter safeguards against larger variations in the EM signal due to the unpredictable activities by the OS.

Threshold θ: The threshold θ helps to distinguish the malicious activities from the normal activities and is chosen using the PDF of the squared prediction error. If the squared prediction error has a bi-modal and disjoint PDF for normal and malicious activity, we can achieve 100% detection with zero false positive by setting the threshold θ between the two PDFs (as in Figure 4.7). However, if the two PDFs overlap, the value of θ is a trade-off between false positives and false negatives. A higher value of θ will lead to lower false positives at the cost of higher false negatives, and vice versa. Note that, in case of zero-day attacks, we don't have prior knowl-

edge about the PDF for the malicious activity. Thus, we set the threshold θ slightly right to the tail of the PDF corresponding to the normal activity.

4.4 Experimental Evaluations

We evaluate the proposed system with several different types of malware on different applications and embedded systems.

4.4.1 Embedded Device with Different Malware Behavior

We implement two types of embedded system malware payloads (DDoS attacks, Ransomware attacks) and a code modification attack (similar to Stuxnet) on an Altera DE-1 prototype board (Cyclone II FPGA with a 50 MHz NIOS II soft-processor). The DDoS attack exploits vulnerabilities such as buffer-overflow to divert the control-flow to send DDoS packets in rapid succession through the devices JTAG port. We also implement a Cryptoviral Ransomware [94] that performs only a single (16-byte) block encryption of AES-128. Intuitively, larger encryption should be easier to detect. Finally, we evaluate a Code Modification attack where the source code has been slightly modified. We added a small (about 10 instructions) to the source code to mimic the behavior of Stuxnet-like malware where the adversary modifies the code to change a critical value based on some conditions.

We inject these malware behavior into three selected applications (*Print Tokens*, *Replace* and *Schedule*) from SIR repository [93]. The system was trained and tested with a disjoint set of user inputs (i.e., the training and testing executions has different user inputs and thus, follow different control-flow paths). Consequently, there were significant variations in execution time for different inputs. For instance, in *Replace*, the shortest

execution lasts only $71\ \mu s$ while the longest one is $4.58\ ms$. Likewise, in *Print Tokens*, the shortest execution is $116\ \mu s$, and the longest execution takes $10.8\ ms$. Similarly, for *Schedule*, the shortest execution is $48\ \mu s$ and the longest execution takes $12.2\ ms$. We used inputs (for both training and testing) that provide high path coverage (using LLVM to find the paths). For example, the *Print Tokens* application has 87 unique acyclic control-flow branches, out of which 83 were executed by the test set. Likewise, the *Replace* application has 96 unique acyclic control-flow branches, out of which 74 were executed during testing. Similarly, the *Schedule* application has 83 unique control-flow branches, and all of them were executed by the test set.

The training and the cross-validation program executions were uncompromised (i.e., without malware), while the testing contained both compromised and uncompromised program executions. For *Print Tokens*, we used 400 training, 45 cross-validation, and 192 testing executions, of which 66 had DDoS, 68 had ransomware, 8 had code modification, while 50 were without malware. Likewise, for *Replace*, we used 458 training, 45 cross-validation, and 188 testing executions. The testing set contained 65 DDoS, 68 Ransomware, 5 code modification malware, and the rest (i.e., 50) were uncompromised. The *Schedule* benchmark had 284 training, 103 cross-validation, and 294 testing examples. The testing set included 67 DDoS, 68 Ransomware, 9 code modification, and 150 executions were without malware.

Figure 4.9 demonstrates our experimental setup. We monitor the device executing these applications using a 2.4-2.5 GHz 18 dBi panel antenna and demodulate the received EM signal using an Agilent MXA N9020A spectrum analyzer. The demodulated signal is then filtered using an anti-

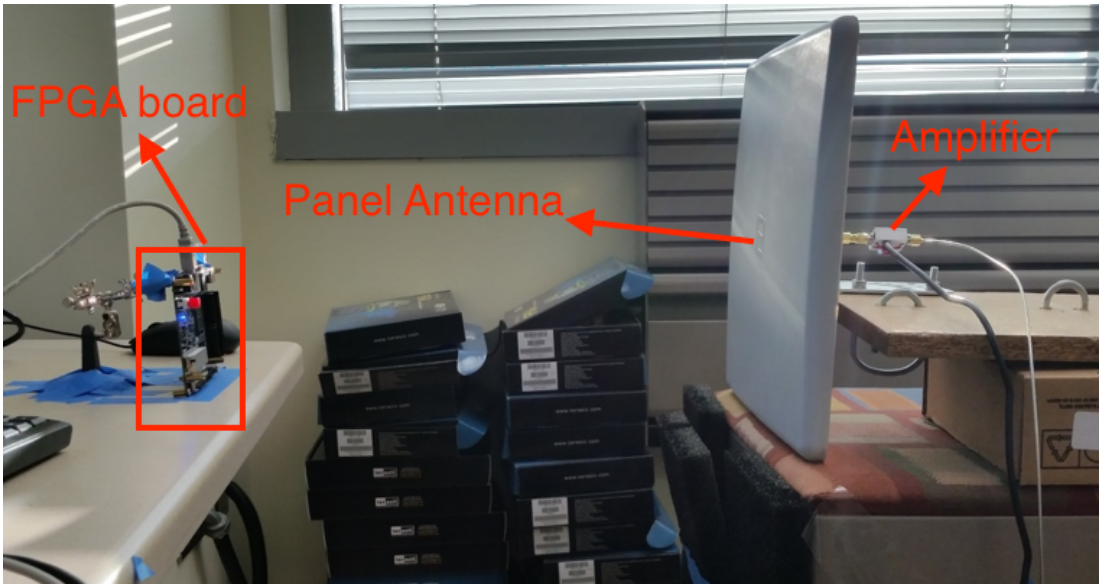


Figure 4.9: Experimental setup of the malware detection system.

aliasing filter with 5 MHz bandwidth and finally sampled at 12.8 MHz sampling rate. The Experimental results demonstrate that the mean squared prediction error for the malicious (i.e., untrained) activity is significantly higher than that of the normal (i.e., trained) activity. This is also shown in Figure 4.7. While the neural network can successfully model and predict the EM signal for trained program activity with low prediction error, the model fails for untrained program activity. As such, any execution of anomalous program activity leads to deviations in the device's EM emanation, which in turn results in higher prediction error. Thus, the system can differentiate between normal and anomalous program activity through the neural network's prediction error.

Table 4.1 demonstrates the performance of the proposed system for detecting different malware activities on different applications. Results show that the system can detect all DDoS and Ransomware without any false positive (AUC = 1.0), and for code modification, the system achieves roughly 0.99 AUC. It should be noted that the execution time for DDoS and

Table 4.1: Detection performance for different malware behavior.

Application	Area Under the Curve (AUC)		
	DDoS	**Ransomware**	**Code Mod.**
Print Tokens	1.0	1.0	1.0
Replace	1.0	1.0	0.99
Schedule	1.0	1.0	0.97

Ransomware is much larger (roughly 25 μs and 150 μs respectively) than that of the code modification attack, which takes up only 5 μs. Hence, code modification is stealthier and harder to detect.

We further evaluate the detection latency of the system. We use a non-causal prediction model (i.e., the neural network exploits both past and future samples to predict the present sample value). This causes a delay of $d = 72$ samples (5.625 μs) in prediction. In addition, the moving average filter introduces a delay of $N = 64$ samples (5 μs). Thus the total system delay is $d + N = 136$ samples (10.625 μs). However, the detection latency will be higher than the system delay due to the time taken for threshold breaching by the anomalous EM pattern. The experimental mean detection latency for DDoS, Ransomware, and code modification are presented in Table 4.2. Both DDoS and code modification are detected in less than 13 μs while Ransomware is detected in 22 μs. In comparison, [34] and [33] has latency greater than 200 μs and 2000 μs respectively.

Table 4.2: Detection latency for different malware behavior.

	DDoS	**Ransomware**	**Code Mod.**
Latency	12.5 μs	22.0 μs	12.5 μs

4.4.2 Robustness against Variations in Antenna Distance

To evaluate the robustness of the system, we trained and tested the system by placing the antenna at different positions. It is reasonable to as-

Table 4.3: Detection performance at different distances.

Distance	Area Under the Curve (AUC)		
	DDoS	**Ransomware**	**Code Mod.**
1m	1.0	1.0	0.99
2m	1.0	1.0	0.99
3m	0.99	1.0	0.97
4m	0.96	0.94	0.71

sume that the system will be trained with a reference device and then deployed to monitor a different target device. As such, the antenna placement and positioning may vary between the training and the monitoring phase. Thus, the detection system must be robust against variations in antenna placements. To evaluate the robustness of the system, we first trained the system from 1 m distance and then used this trained system to monitor the target device from four different distances (1 m, 2 m, 3 m, and 4 m). Table 4.3 shows that the system is robust against variations in antenna distance. In addition, the system demonstrates excellent performance from up to 3 m distance. Further distance causes some degradation in system performance due to the lower Signal-to-Noise Ratio (SNR) at a higher distance. Note that our framework is not limited by distance, and higher distance coverage can be achieved by using higher gain antennas (e.g., [109]).

4.4.3 Robustness against Noise and Interference

We further evaluate the robustness of the system against environmental noise by applying Additive White Gaussian Noise (AWGN) to the monitored signal. Any practical monitoring system should be able to detect security threats under a potentially noisy environment. Thus, we evaluate the performance of the detection system at different SNR by applying AWGN to the monitored signal. Table 4.4 shows that the system is robust against

Table 4.4: Detection performance at different Signal-to-Noise Ratio.

SNR	Area Under the Curve (AUC)		
	DDoS	**Ransomware**	**Code Mod.**
30 dB	1.0	1.0	0.99
20 dB	1.0	1.0	0.98
10 dB	1.0	1.0	0.95
5 dB	0.85	0.95	0.71

noise and has an excellent detection performance even at an SNR as low as 10 dB.

In addition, the system is inherently robust against any EM interference outside its monitored bandwidth. As described in Section 2.3.2, the anti-aliasing filter used during the analog-to-digital conversion nullifies any signal with frequencies beyond $f_c \pm B$. Here, f_c is the clock frequency of the monitored device, and $2B$ is the monitored bandwidth. Thus, any EM interference outside the monitored bandwidth does not influence the detection performance.

4.4.4 Attack on IoT Device

We implement three different malicious activities (e.g., code injection, DDoS, and Ransomware) on an IoT device (A13-OLinuXino board with 1 GHz Cortex A8 ARM processor and Debian Linux OS). We inject these malicious behavior into two selected applications (*basic math* and *bit count*) from MiBENCH [110]. First, we implement a buffer overflow attack to inject *shellcode* into the application. Next, we port a DDoS *bot* in a selected location of the application. The DDoS *bot* sends 100 TCP SYN packets and then resumes to normal program activity. Finally, we implement a Ransomware prototype that performs AES 128 encryption.

We monitored the emanated EM signal with a small magnetic probe placed 5 cm away from the system using a commercially available software-

Table 4.5: Detection performance for monitoring IoT device.

Application	Area Under the Curve (AUC)		
	Code Inj.	**DDoS**	**Ransomware**
Basic Math	1.0	1.0	1.0
Bit Count	1.0	1.0	1.0

defined radio (Ettus Research B200-mini) with a bandwidth of 40 MHz centered at the clock frequency (1 GHz) of the device. The collected signal was then demodulated, digitized, down-sampled to 10 MHz sampling rate, and finally processed through the proposed neural network framework. For each application, we trained the system with 25 uncompromised (malware-free) executions. Next, we test the system with 100 executions (25 malware-free, 25 with code injection, 25 with DDoS, and 25 with Ransomware). Experimental evaluations (in Table 4.5) show that the system detects all malicious activity without any false positive.

We used the same neural network architecture and parameters (e.g., D=128 and k = 8) throughout all experiments. However, we exploited a higher order moving average filter (N=1024) to avoid false positives due to transient activities by the OS. Consequently, the detection latency of the system was higher (roughly $120\ \mu s$), which is still considerably lower than [34] and [33] ($200\ \mu s$ and $2000\ \mu s$ respectively). Note that, [33] used a similar experimental setup (e.g., the same benchmark applications executed on the same device with similar code injection attacks). However, [34] monitored a PLC - a simpler device (e.g., slower clock speed and does not have an OS). Intuitively, it should be easier to model EM emanation from a simpler device (e.g., in the absence of unpredictable OS activities), and thus should lead to lower detection latency.

4.4.5 Attack on Medical Cyber-Physical System

We further evaluate the system by implementing malicious attacks on a medical CPS called SyringePump. A SyringePump is a medical device that can dispense or withdraw a precise amount of fluid or medicine [111]. A SyringePump has three main components, a syringe filled with medicine, an actuator (typically a stepper motor), and a control unit that receives user inputs and controls the actuator accordingly.

To evaluate the robustness of the proposed malware detection system, we implement an Open Source SyringePump [112] with two different devices:

1) TS-7250 Board (200 MHz Cirrus EP9302 ARM9 CPU with a Debian Linux OS), and

2) A13-OLinuXino Single-Board-Computer (1 GHz ARM Cortex A8 processor with a Debian Linux OS).

We exploit a buffer overflow vulnerability in the serialRead() function to hijack the control-flow and call MoveSyringe() function to dispense or withdraw an unwanted amount of fluid. This is an example of a code-reuse attack where the attacker repurposes existing code to perform unwanted action. As the attacker executes existing code, albeit, in an undesired way, a code-reuse attack can be harder to detect. Any failure to administer medication at an appropriate dosage can have serious consequences for the patient. Thus, this attack poses a critical threat to the integrity of the SyringePump. For monitoring, we place a small magnetic probe 5 cm away from the system and record and demodulate the signal using a commercially available software-defined radio (Ettus Research B200-mini).

We train the system with 25 executions and test it with 50 executions, out of which half were compromised with malware. Experimental results

Table 4.6: Detection performance for monitoring SyringePump.

	TS-7250 Board	**A13-OLinuXino**
AUC	1.0	1.0

(in Table 4.6) show that the system achieves excellent performance and detects all malicious activity without any false positive.

4.5 Summary

In this chapter, we presented a novel framework for malware detection in critical and high-assurance embedded and cyber-physical systems using EM side-channel signal analysis. The system models device's EM emanation with a multilayer perceptron (MLP) and detects anomalous or malicious program activity through deviations in the EM fingerprint. The system is trained with EM signal from an uncompromised reference device and can predict EM emanation for normal (i.e., trained) program activity. However, whenever the monitored device performs any malicious (i.e., untrained) program activity, the trained neural network model fails and results in high prediction error. We then detect this deviation in prediction error and report anomalous activity. The system does not require any knowledge about the nature of the attack or its malware signature, thus ensures protection against zero-day attacks. In addition, the system can provide non-intrusive and remote monitoring (without any physical access to the device), and does not require any modification to the monitored system. Neither does it impose any overhead on the monitored device. The detection system can train its model by observing the device's EM emanation and does not require any access to the source code or the control-flow graph of the monitored system. We demonstrated the effectiveness of the system with several key malware behavior (DDoS, Ransomware, and Code

Modification), which the system could detect with excellent accuracy (AUC $\approx$ 0.99) from up to 4 m away. The system was also able to detect attacks on an IoT device and a medical CPS with 100% accuracy.

CHAPTER 5
PROGRAM TRACING THROUGH ELECTROMAGNETIC SIDE-CHANNEL ANALYSIS

5.1 Overview

In this chapter, we present P-TESLA (Program-Tracing through Electromagnetic Side-channeL Analysis), a novel framework for zero-overhead program execution tracing. P-TESLA leverages the device's electromagnetic (EM) side-channel signals for basic-block-granularity program execution tracing. P-TESLA is completely non-invasive and does not require any resource or any modification of the monitored device. Thus, P-TESLA is especially suitable for monitoring resource-constrained devices such as embedded devices and the Internet of Things (IoT) devices.

To reconstruct program execution traces, P-TESLA has to overcome the following challenges: 1) train a signal emanation model that associates signal patterns (or signatures) with code segments or subpaths, and 2) represent the test signal using such signal patterns to reconstruct the program execution path. Specifically, we use a two-step training process that exploits instrumented training to annotate the uninstrumented training signals, and identify which signal snippets correspond to which code segments. We also propose a novel signal matching technique that efficiently establishes a correspondence between the test signal and the training signals, and exploit this signal correspondence to reconstruct the program execution path.

The main contributions of this chapter are:

- P-TESLA - a novel framework for zero-overhead and non-intrusive program tracing.
- A training process that exploits instrumented executions to annotate uninstrumented training signals.
- An efficient signal matching algorithm that establishes a correspondence between the training and the test signals, and reconstructs the execution path based on the signal correspondence.
- Empirical evaluations that demonstrate that (1) P-TESLA achieves high accuracy and the predicted timestamps are highly precise (2) P-TESLA can monitor devices with fast processors and operating systems, and (3) P-TESLA is able to monitor devices from 1 m distance.

The rest of the chapter is organized as follows: Section 5.2 details our framework for program execution tracing, Section 5.3 presents the experimental evaluations, and finally, Section 5.4 presents the summary.

5.2 Program-Tracing through Electromagnetic Side-ChanneL Analysis

P-TESLA provides non-intrusive and zero-overhead program tracing by monitoring the EM side-channel signal. A high-level overview of P-TESLA is demonstrated in Figure 5.1. In the training phase, P-TESLA first executes an instrumented version of the program and records the corresponding EM emanations. The instrumented program also outputs a marker sequence and their execution timestamps that indicate the program execution path. P-TESLA next executes an uninstrumented (unaltered) program with the same program input and compares the uninstrumented EM signal with the instrumented EM signal to map the uninstrumented signal fragments to the underlying program subpaths. We call this mapping

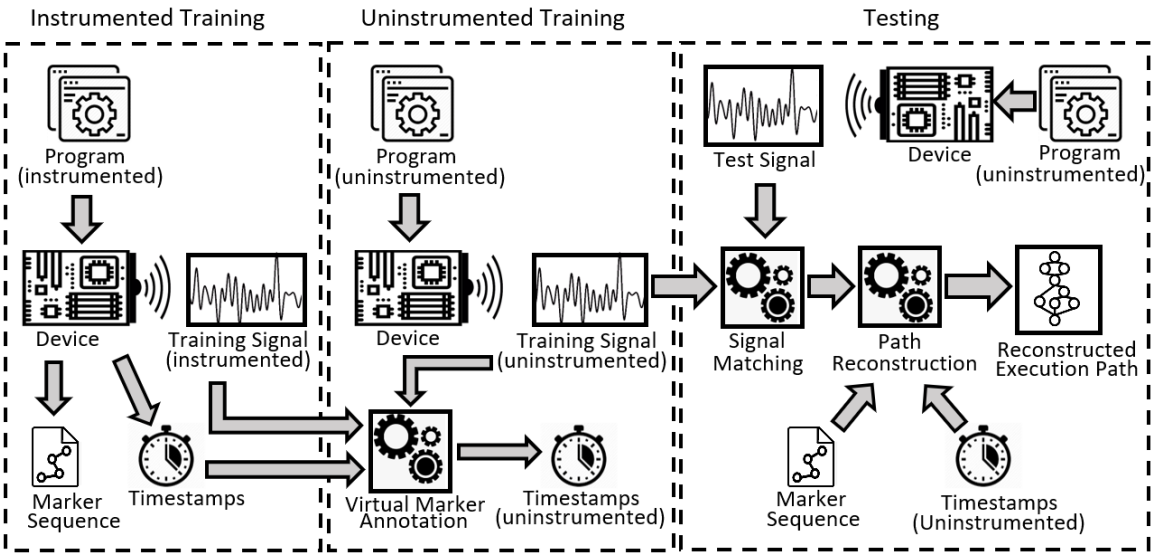


Figure 5.1: Overview of the P-TESLA framework.

Virtual Marker Annotation, as it mimics the markers, however, without any code instrumentation. This process exploits the fact that given the same program input, the instrumented and the uninstrumented programs follow the same program path, and thus follow the same marker sequence. This process also exploits instrumented markers and timestamps for efficient and precise signal mapping. Next, in the testing phase, P-TESLA monitors the EM side-channel signal caused by the execution of the vanilla (i.e., uninstrumented and unaltered) version of the program. P-TESLA then matches the test signal with the training signals, and exploits *virtual markers* to reconstruct the program execution path. In the following sections, we explain the different steps of P-TESLA in detail.

5.2.1 Signal Preprocessing: Amplitude Demodulation

To monitor the program execution, P-TESLA first receives the emanated EM signal through an antenna, performs amplitude demodulation of the received signal, and then digitizes the demodulated analog signal using an analog-to-digital converter. The digitized signal is next scale normalized

before any further signal analysis. These preprocessing steps are applied to both training and testing phases.

Researchers have demonstrated that embedded devices emanate amplitude modulated signals at the device's clock frequency [89, 88]. At each processor cycle, the CPU executes instructions, and thus, changes the states of its internal digital circuits (i.e., switches on and off). This causes an instruction dependent current at the CPU clock frequency. Here, the CPU clock acts as the carrier, whose amplitude (i.e., the pulse shape) is modulated by the variations of the executed instructions [113]. As this current flows within the processor, and through the device's printed circuit board (PCB), the device acts as an unintentional and inefficient antenna and emanates amplitude modulated EM signal [91]. Thus, to monitor program execution, we demodulate the received signal $r(t)$ at the CPU clock frequency f_c.

$$x_a(t) = |r(t) \times e^{j2\pi f_c t}| \tag{5.1}$$

Here, $x_a(t)$ is the amplitude demodulated analog signal and t denotes the time. The demodulated signal $x_a(t)$ is then passed through an anti-aliasing filter with bandwidth B, and sampled at a sampling period T_s.

$$x_d(n) = x_a(nT_s) \tag{5.2}$$

Here, $x_d(n)$ denotes the sampled signal at sample index n. The anti-aliasing filter cancels unwanted signals with frequencies beyond $f_c \pm B$. Note that, the sampling period T_s is determined by the well known Nyquist criterion $\frac{1}{T_s} > 2B$. Next, we scale normalize $x_d(n)$:

$$x(n) = \frac{x_d(n)}{max(x_d(n))} \tag{5.3}$$

Scale normalization ensures that the system is robust against changes in amplitude of the EM signals (e.g., due to changes in the antenna's distance, position, etc.). The scale normalized signal $x(n)$ is then used for further signal analysis by P-TESLA in the training and the testing phases.

5.2.2 Instrumented Training

P-TESLA is first trained with instrumented program executions and their corresponding EM emanations. We execute an instrumented version of the program, in which the source code is instrumented by inserting markers (i.e., special probe functions) at selected program locations. Each marker has a unique identification number (ID) that identifies its position in the program's control-flow-graph (CFG). The marker function execution records the marker ID along with the execution timestamp. Thus, the markers perform as program execution checkpoints that partition the CFG into smaller code-segments, which we refer to as marker-to-marker code-segments or subpaths.

We insert these markers in strategic program locations. The marker insertion is dictated by the following criteria: (1) any program execution control-flow path must be uniquely and unambiguously represented by a sequence of marker-to-marker subpaths, and (2) all marker-to-marker subpaths must be acyclic and intra-procedural. Based on these criteria, we inserted markers in the following code locations: entry and exit nodes of functions, loop heads, and target nodes of go-to statements.

CFG partitioning helps to provide training coverage for program execution. Specifically, any practical program has a large number of feasible program execution paths. In fact, due to cyclic paths in the CFG, programs can have an infinite number of unique execution paths. Hence, it is neither

practical nor possible to provide training for all unique execution paths for any practical program. However, the markers enable us to represent any execution path as a concatenation of marker-to-marker subpaths. Thus, instead of providing training for all unique execution paths, we provide training that covers all marker-to-marker subpaths. Note that the number of marker-to-marker subpaths is limited and can be exercised using a relatively fewer number of strategic executions.

The markers also enable us to annotate the monitored EM signal. The marker execution timestamps help to establish a correspondence between executed code segments (i.e., marker-to-marker subpaths) and the signal fragments they generate. It is important to emphasize that while the markers provide an abstract partitioning, the program execution (for both training and testing) follows a single contiguous trace (from the program's start to end), and generates a continuous EM signal. Consequently, it is not visually identifiable that which marker-to-marker subpath generated which part of the emanated signal. So, we use the marker execution timestamps to annotate the start and the end of each marker-to-marker subpath in the monitored EM signal.

At the beginning of the program execution, we reset the processor's Time Stamp Counter (TSC) to zero. The markers record the TSC values as timestamps, which then indicate the time-interval (in clock-cycles) from the program's start. We convert these timestamps to sample-index using the following equation:

$$n = round(\frac{t \times f_s}{f_c}) \tag{5.4}$$

Here, n indicates the sample-index corresponding to the timestamp t, f_s is the sampling rate of the monitored signal, and f_c is the clock frequency of the monitored CPU. The rounding operation ensures that the resultant n

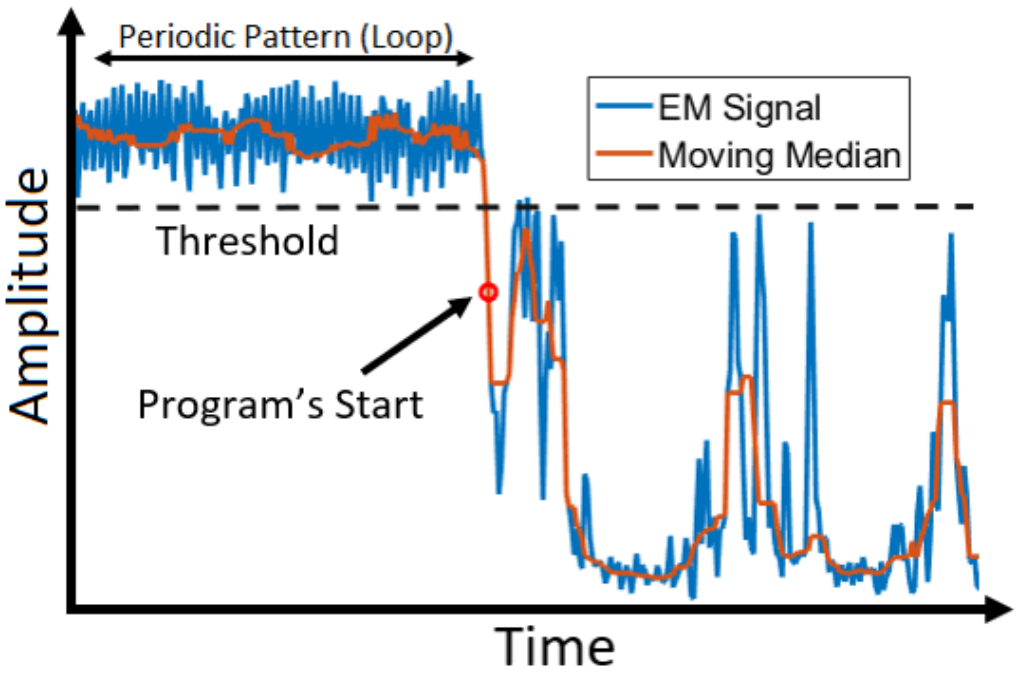


Figure 5.2: Automatic detection of program's start: the end of the periodic pattern (for-loop) indicates the program's start. We identify this when the moving average of the EM signal drops below the threshold.

is an integer value.

We also identify the program's start (i.e., sample-index $n = 0$) in the signal. To facilitate the automatic detection of the program's start (demonstrated in Figure 5.2), we insert a for-loop just before the beginning of the program execution. The for-loop executes a repetitive activity (e.g., increments a loop counter variable) and generates a periodic and identifiable signal pattern. The end of this periodic pattern indicates the end of the for-loop (i.e., the start of the program). Furthermore, at the beginning of the program execution, the program is loaded into the system memory. This leads to memory access, which in turn stalls the processor and causes a dip in the signal amplitude [81]. We identify this transition (from the end of for-loop to the beginning of program execution) when the moving median of the signal drops below a predefined threshold (as shown in Figure 5.2). This acts as the reference point (i.e., sample-index $n = 0$). All markers are

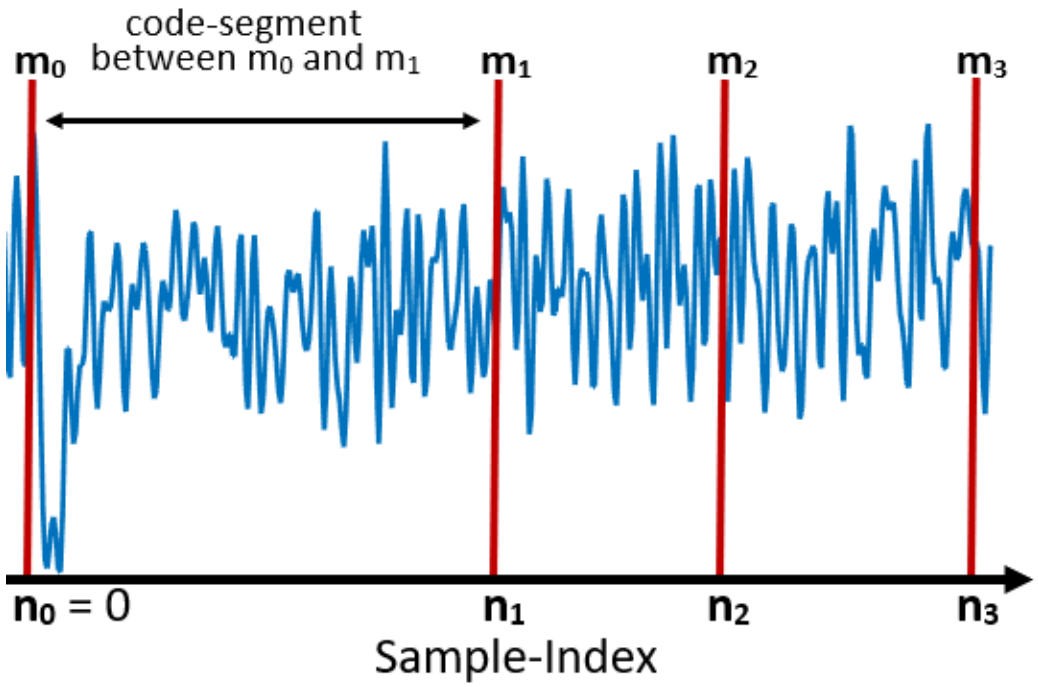


Figure 5.3: Markers (vertical red lines) are placed on the signal according to their execution timestamps. The signal snippet between two consecutive markers corresponds to the EM emanation from the marker-to-marker code-segment.

then annotated according to their sample-indices.

Figure 5.3 demonstrates an annotated instrumented signal with each marker represented by a vertical red line. Markers m_0, m_1, m_2, and m_3 are placed at sample-index n_0, n_1, n_2, and n_3 respectively. The marker ID sequence (e.g., m_0, m_1, m_2, ...) indicates the program execution path, while execution timestamps or sample-index sequence (e.g., n_0, n_1, n_2, ...) identifies the start/end of the marker-to-marker code-segments. Thus, marker annotation establishes a correspondence between code-segments and emanated EM signal snippets. For instance, in Figure 5.3, the signal snippet between sample-index n_0 and n_1 corresponds to the execution of the code-segment or subpath between marker m_0 and m_1.

While the instrumented training helps us to partition the CFG and to annotate the signal, the instrumentation alters the original signal emana-

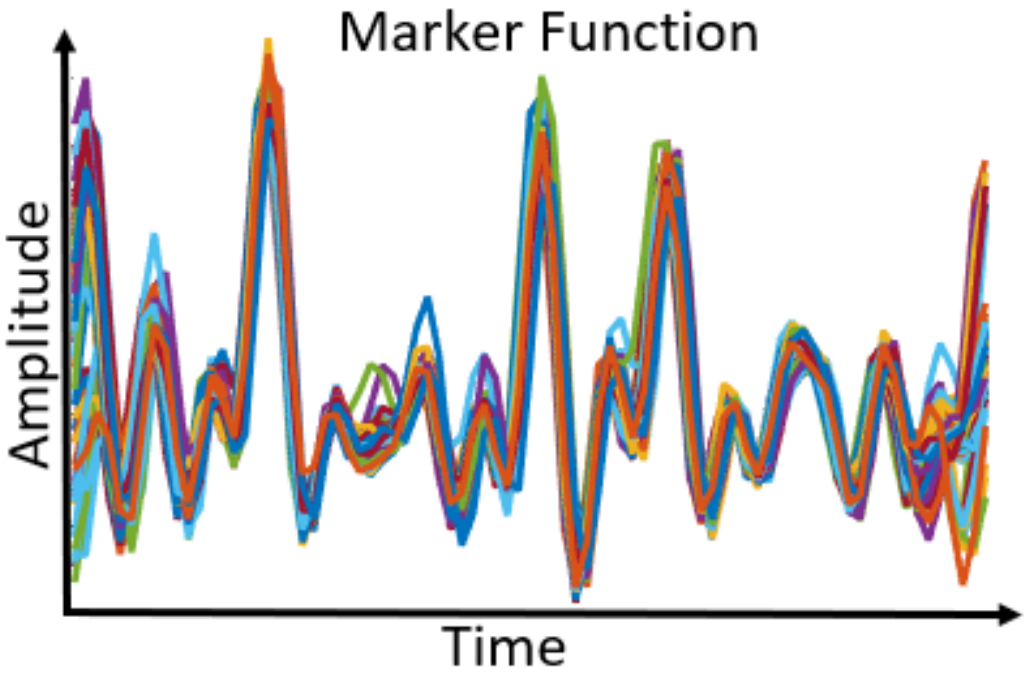


Figure 5.4: EM signals corresponding to marker function execution.

tion patterns. Thus, the instrumented training signals and corresponding signal emanation models cannot be used in the testing phase, in which the device executes a vanilla version (i.e., unaltered and uninstrumented) program. Specifically, the instrumentation adds overheads to the original program (i.e., function calls that record marker ID and execution timestamps). The execution of these overhead codes (i.e., marker functions) requires additional computation (and computational time), and in turn, causes extraneous EM emanations that are irrelevant to the original program.

To evaluate the impact of instrumentation, we investigate the EM signature of the marker functions. We identify the marker functions in the instrumented signal using their timestamps. We then crop out and compare these signal snippets. Figure 5.4 overlays 100 signal snippets corresponding to the marker function execution. We observe that the marker functions emanate very similar signal patterns. This is expected as the

marker functions execute the same code segment. However, the beginning and the end of the marker signals demonstrate a marked variability. This is due to the microprocessor's instruction pipeline architecture that overlaps multiple instructions during execution. Thus, the processor's EM emanation at any instance depends on all instructions that are moving through different stages of the pipeline, rather than just one single instruction. Consequently, the execution of the same marker function may demonstrate signal variability towards the beginning and the end of the function call depending on the variations in the preceding and the following code segments (i.e., other instructions in the pipeline).

Likewise, the marker functions themselves also affect the EM emanations of the adjacent code segments. In the instrumented execution, all marker-to-marker code-segments are separated by marker functions. As such, the EM emanation patterns from the code-segments are altered at the boundaries due to the "cross-over" effect from the marker functions. For larger code segments (e.g., consisting of a few hundred instructions), the duration of the emanated signal is much larger compared to the altered boundaries. Thus, the impact of instrumentation is trivial. However, for smaller code segments such as basic blocks consisting of only a few instructions, instrumentation can alter the overall signal emanation pattern significantly. Thus, cropping out the marker signal-snippets from the instrumented signals would not replicate the uninstrumented signals. Therefore, we exploit uninstrumented executions to create a better signal emanation model.

5.2.3 Uninstrumented Training

P-TESLA is next trained with uninstrumented program executions. However, before we can use the uninstrumented training for program execution monitoring, we must first annotate the signal. Unlike the instrumented executions, the uninstrumented executions do not have markers or timestamps. Thus, we cannot directly annotate or identify which signal snippet corresponds to which code segment. Instead, we compare uninstrumented execution with the instrumented execution to identify and demarcate the marker-to-marker code segments in the signal. We call these demarcations "virtual markers" as they play the same role as the marker functions, albeit, without adding any overhead code or altering the original program or its signal emanation patterns.

Virtual Marker Annotation: In the instrumented execution, code segments are separated by marker functions. Each marker function execution records a pair of information m and t, where m represents the marker ID that indicates the execution point in the CFG, and t is the execution timestamp. We then convert the timestamp t to its equivalent sample-index n (using equation 5.4). If the program executes k marker-to-marker code-segments, the instrumented execution records a sequence of $k+1$ markers (including the starting and the ending markers). Thus, instrumented execution outputs a maker ID sequence $M = \{m_0, m_1, m_2, ..., m_k\}$ and corresponding sample-index sequence $N = \{n_0, n_1, n_2, ..., n_k\}$, with M uniquely identifying the program execution path, and N indicating which signal snippet corresponds to which code segment. Thus, the task of virtual marker annotation is to generate marker ID sequence M' and sample-index sequence N' for the uninsrumented training signal, without actually

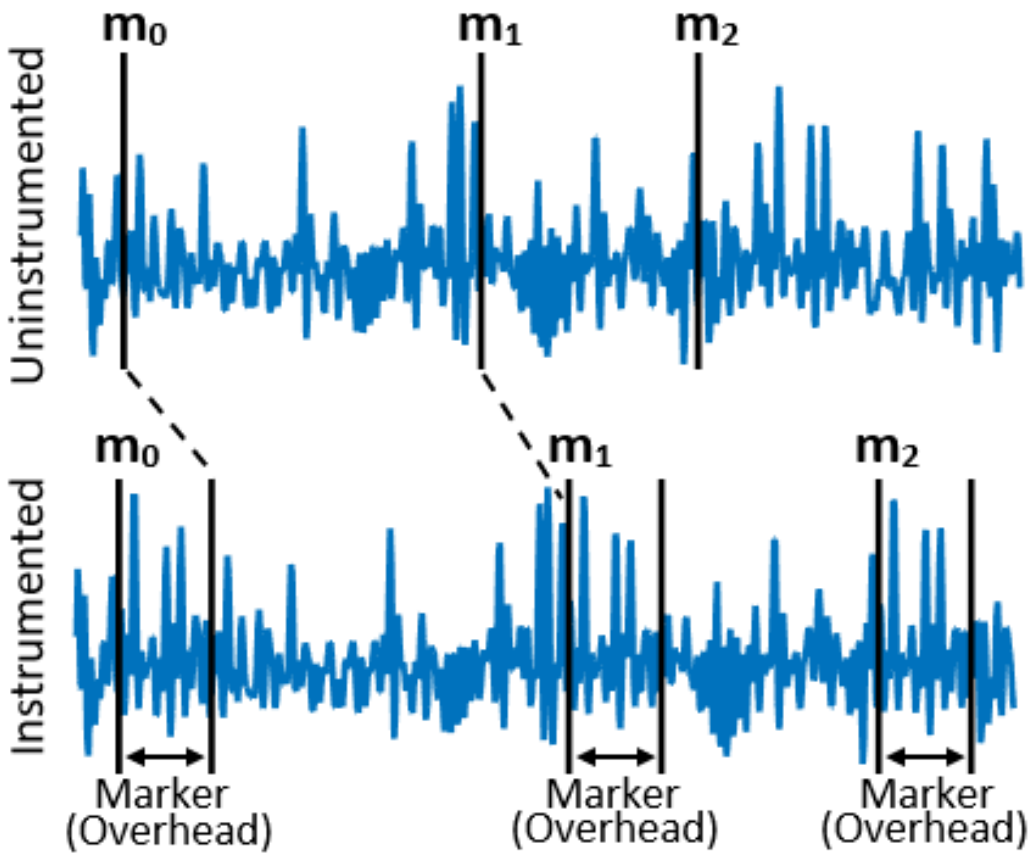


Figure 5.5: EM signals corresponding to the uninstrumented (top) and the instrumented (bottom) program executions. The dotted lines indicate the correspondence between the uninstrumented and the instrumented signal.

using instrumentation or marker functions.

To annotate the virtual markers, we execute the instrumented and the uninstrumented programs with the same input. Thus, the executions follow identical paths through the CFG (i.e., execute the same marker-to-marker code segments in the same order). This ensures the marker ID sequence is identical for instrumented and uninstrumented executions (i.e., $M' = M$). However, due to the overhead computations (i.e., marker functions), the timestamps or sample-indices for the virtual markers are significantly different (i.e., $N' \neq N$).

To estimate the virtual marker sample-index sequence N', we compare

the uninstrumented and instrumented EM signals (Figure 5.5). We notice that the execution of the same code segment (e.g., m_0-m_1-m_2) requires more computational time in the instrumented version due to the marker function overheads. Thus, we estimate the sample-indices for the virtual markers by adjusting for the overhead computational time using the following equation:

$$n'_i = n'_{i-1} + (n_i - n_{i-1} - n_{oh}) \quad \text{for } i \in \{1, \ldots, k\} \tag{5.5}$$

Here, n'_{i-1} and n'_i indicate the sample-indices for the $(i-1)$-th and i-th virtual marker in the uninstrumented signal, n_{i-1} and n_i indicate the sample-indices for the $(i-1)$-th and i-th marker in the instrumented signal, and n_{oh} is the overhead computational time (in samples) for the marker function. Thus, $(n_i - n_{i-1} - n_{oh})$ is the overhead-subtracted execution time for the i-th code segment. Note that, sample-index $n'_0 = 0$ (indicating the starting point of the program), and we iteratively estimate n'_1, n'_2, ..., n'_k.

While Equation 5.5 gives a good initial estimation for the virtual marker annotation, it does not account for the execution-to-execution hardware variabilities such as cache hits or misses that may lead to variabilities in computational time. To mitigate this issue, we fine-tune the initial sample-index estimations by matching the uninstrumented signal with its instrumented counterpart. First, we identify the signal snippet corresponding to a given code segment in the instrumented signal using its timestamps. Let $x(n)$ be the instrumented signal with n indicating its sample-index. Thus, the signal snippet between sample index $n = n_{i-1}$ and $n = n_i$ corresponds to the i-th code segment (i.e., the subpath between markers m_{i-1} and m_i). We then exclude or crop-out the first n_{oh} samples from this sig-

Algorithm 1: Virtual marker annotation.

Input: $x(n)$, $y(n')$, $N = \{n_1, n_2, \ldots, n_k\}$
Output: $N' = \{n'_1, n'_2, \ldots, n'_k\}$
initialization: set $n'_0 = 0$;
for $i := 1$ ***to*** k **do**
    Estimate n'_i (Eq. 5.5);
    Find best match (Eq. 5.6);
    Update n'_i (Eq. 5.7);

nal snippet as they correspond to the marker function **not** the original code segment. In Figure 5.5, the dotted lines indicate the correspondence between the uninstrumented and instrumented signals. Thus, this overhead-subtracted signal snippet $s_i(n)$ acts as the EM signature or template for the code segment. We search for this signal template by sliding it across the uninstrumented signal $y(n')$. We limit our search within $\pm d$ samples of the initial estimations (i.e., between sample-index $n' = n'_{i-1} - d$ to $n' = n'_i + d$). This makes the search computationally efficient, and also helps to avoid false signal matches. At each search position, we compute the Euclidean distance between the template and the uninstrumented signal. We then choose the least Euclidean distance match for updating the initial estimations.

$$\hat{l} = \arg\min_{l} e(l) \quad \text{for } l \in [-d, d] \tag{5.6}$$

Here, $e(l)$ is the Euclidean distance between the template $s_i(n)$ and the uninstrumented signal $y(n')$, and l indicates the shift from the initial estimated n'_i. Thus, $\hat{l}$ is the shift corresponding to the best match. Finally, we update initial estimated n'_i using the following equation.

$$n'_i := n'_i + \hat{l} \tag{5.7}$$

This iterative process is depicted in Algorithm 1.

5.2.4 Program Execution Monitoring

To reconstruct the program execution path, P-TESLA compares the device's EM emanation with the (uninstrumented) training signals and predicts the control-flow execution path. The path prediction involves two steps. In step 1, we match the monitored signal with the training signals to establish a signal correspondence. In step 2, we exploit this signal correspondence to predict the program execution path by using the training signal annotations (i.e., the virtual markers). We discuss these steps with further details in the following paragraphs.

Signal Matching: To establish a signal correspondence, we match fixed-length windows from the monitored signal against the training signals, and then adjust the window-size according to the signal similarities. The signal matching process is demonstrated in Figure 5.6. First, we extract a fixed-length initial window W of size L from the monitored signal. We then slide W across all training signals to find the best (i.e., the least Euclidean distance) match. This establishes a window-to-window signal correspondence (shown with a dashed arrow in Figure 5.6). Next, we compare the samples that follow these windows. In Figure 5.6, the initial window and its subsequent samples are overlaid on the matched window and its subsequent samples using red dots. We then iteratively extend the signal correspondence as long as the overlaid monitored signal is similar to the underlying training signal. Specifically, in each iteration, we compare the D subsequent samples and compute the sample-to-sample squared difference. If the mean squared difference is below a predefined threshold θ, we update the matched window size: $L := L + D$, and keep comparing the next D samples. Otherwise, we terminate the window extension process. We

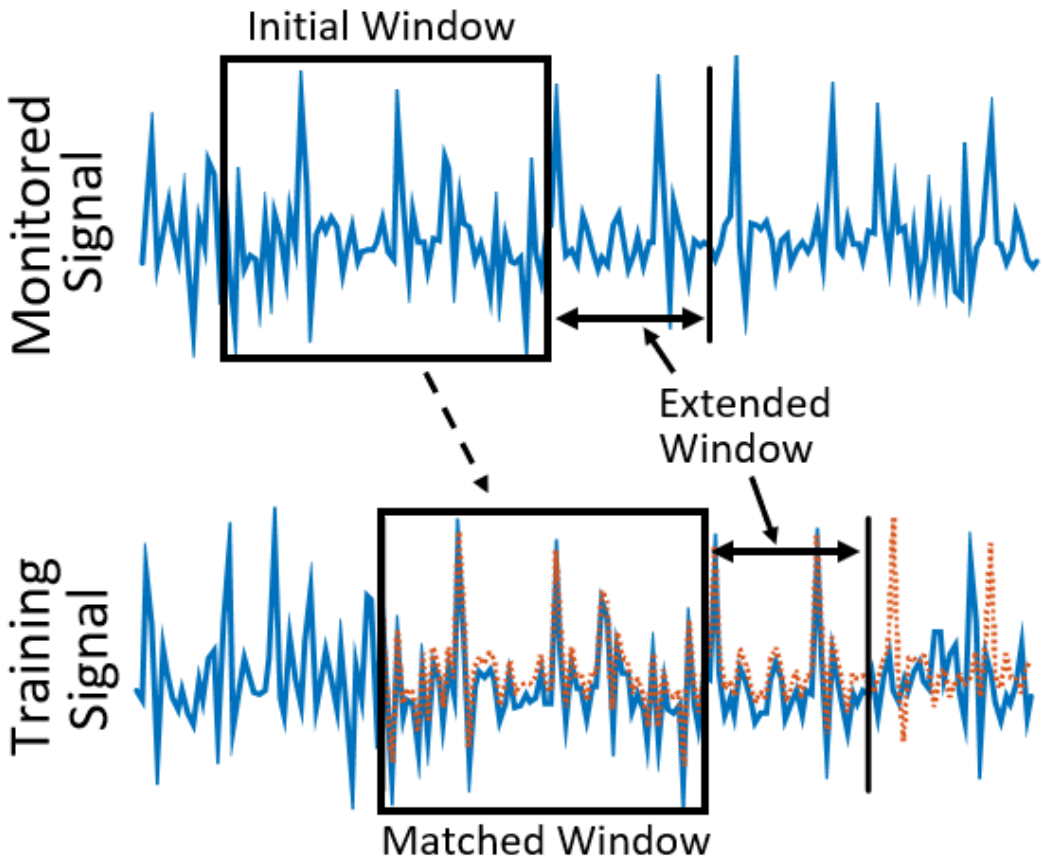
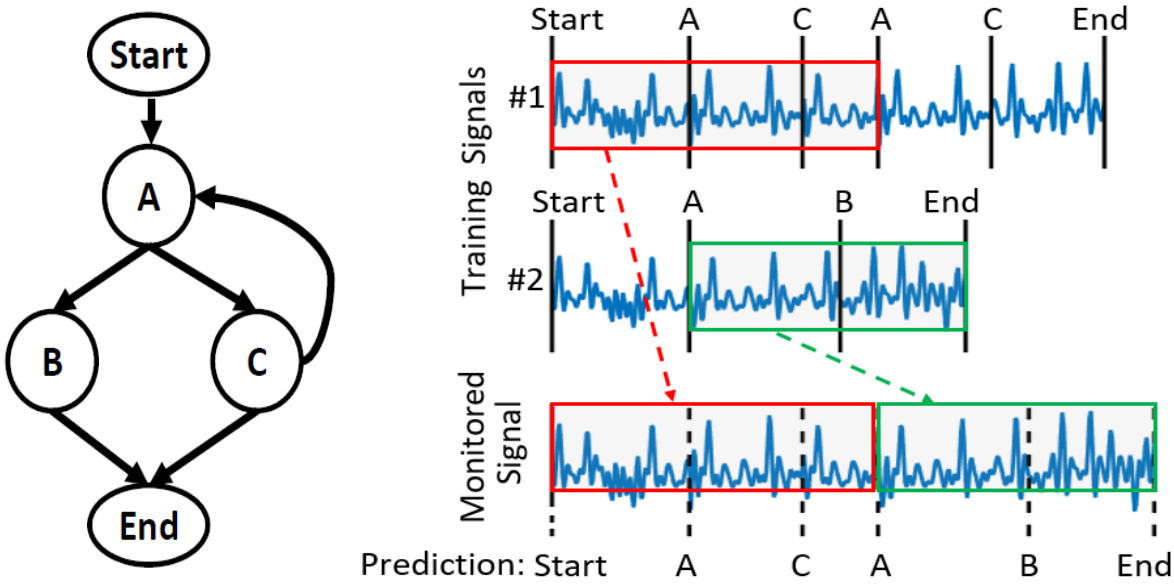


Figure 5.6: Signal matching process: the dashed arrow indicates the correspondence between fixed-length windows in monitored and training signals. Window size is extended based on signal similarities, up to the point where the training signal (blue line) starts to deviate significantly from the monitored signal (overlaid red dots).

then again extract the next unmatched window from the monitored signal, match it across all training signals, and adjust the window-size. This process goes on until we establish signal correspondence for the entire monitored signal.

This approach for signal matching is computationally more efficient than that of multiscale signal matching, in which multiple windows of different sizes are simultaneously matched against the training signals. In contrast, we initiate the search using a small fixed-sized window, and then gradually extend the window size. Furthermore, the time complexity



(a) Control-flow graph (b) Program execution path reconstruction using signal correspondence.

Figure 5.7: Execution path reconstruction exploiting signal correspondence between training and test signals.

for the window search is directly proportional to the window size L. Thus, a smaller window leads to a faster search. However, if the window is too small, the match becomes unreliable. Therefore, in our experiments, we choose $L = 64$. In addition, smaller values for D enable finer adjustment of the window size. However, too small a value for D may lead to early termination of the window extension due to a few noisy samples. In our experiments, we use $D = 8$.

Path Reconstruction: We next exploit the correspondence between the monitored and the training signals to reconstruct the execution path. Figure 5.7 demonstrates the path reconstruction process with a simplified example. On the left (Figure 5.7a), we have the program CFG where the nodes represent the markers, and the edges represent the marker-

to-marker subpaths. The training signals and the monitored signal are shown on the right (Figure 5.7b). The (virtual) markers are annotated on the training signals with vertical black lines and indicate that training signal 1 corresponds to the program path $Start - A - C - A - C - End$, while training signal 2 corresponds to $Start - A - B - End$. Note that, for this simple CFG, these two training executions are sufficient to provide coverage for all marker-to-marker subpaths (i.e., edges on the graph). However, most applications often require a large number of executions (e.g., hundreds or even thousands) for high code coverage.

Furthermore, we indicate the correspondence between the monitored and the training signals using color-matched windows and dashed arrows. For instance, the red windows in the training and the monitored signals demonstrate similar signal patterns, and so do the green windows. This signal correspondence enables us to reconstruct the monitored signal by concatenating matched-windows (e.g., red and green windows) from different training signals. More importantly, the signal similarity or correspondence implies that the matched windows correspond to the same program subpath. Thus, we reconstruct the program execution path for the monitored signal by concatenating the program subpaths corresponding to the matched training windows. For instance, the red window (in training signal 1) corresponds to the program subpath $Start - A - C - A$, and the green window (in training signal 2) corresponds to the program subpath $A - B - End$. Therefore, we concatenate these subpaths to reconstruct the execution path. In Figure 5.7, the reconstructed execution path ($Start - A - C - A - B - End$) is indicated with dashed vertical lines.

5.3 Experimental Evaluations

We evaluate P-TESLA by monitoring two different devices executing three different benchmark applications. The evaluation metrics, the benchmark applications, and the experimental results are discussed in the following sections.

5.3.1 Evaluation Metrics

To evaluate P-TESLA, we compute the edit distance between the actual execution path and the reconstructed execution path. Specifically, we use Levenshtein distance [114] that computes the minimum number of edits (insertions, deletions, or substitutions) required to change the reconstructed marker sequence to the actual marker sequence. We then compute the path reconstruction accuracy using the following equation.

$$\text{Accuracy} = 1 - \frac{\text{Edit Distance}}{\text{Length of Actual Marker Sequence}} \tag{5.8}$$

We further compare the actual and the reconstructed timestamps. Specifically, we compute and report the absolute timing difference between the actual and the reconstructed markers. Note that, the edits are excluded from this comparison, as there is no timestamp for the edited (e.g., inserted or deleted) markers.

5.3.2 Benchmark Applications

We selected 3 benchmark applications (*Print Tokens*, *Replace*, and *Schedule*) from the SIR repository [93]. These applications are commonly used to evaluate techniques that analyze program execution. The size matrix

Table 5.1: Benchmark applications statistics.

Benchmark	LOC	Basic Blocks
Print Tokens	464	178
Replace	495	245
Schedule	579	175

for the benchmark applications is shown in Table 5.1.

Moreover, these applications have many inputs, each taking a unique execution path through the CFG. We used disjoint sets of inputs for training and testing. For each application, we randomly selected 500 inputs for training, and 100 for testing. Table 5.2 summarizes training-testing split.

Table 5.2: Training and testing executions.

Benchmark	Training	Testing
Print Tokens	500	100
Replace	500	100
Schedule	500	100

We evaluate P-TESLA by executing these applications on two different devices: 1) FPGA device and 2) IoT device.

5.3.3 FPGA Device Monitoring

First, we monitored an Altera DE-1 prototype (Cyclone II FPGA) board. This device has a 50 MHz NIOS II soft-processor. We placed a magnetic probe near the device to collect the EM side-channel signal. We then used an Agilent MXA N9020A spectrum analyzer to observe and demodulate the EM emanations. The demodulated signal is next passed through an anti-aliasing filter with 5 MHz bandwidth. Finally, we sampled the filtered

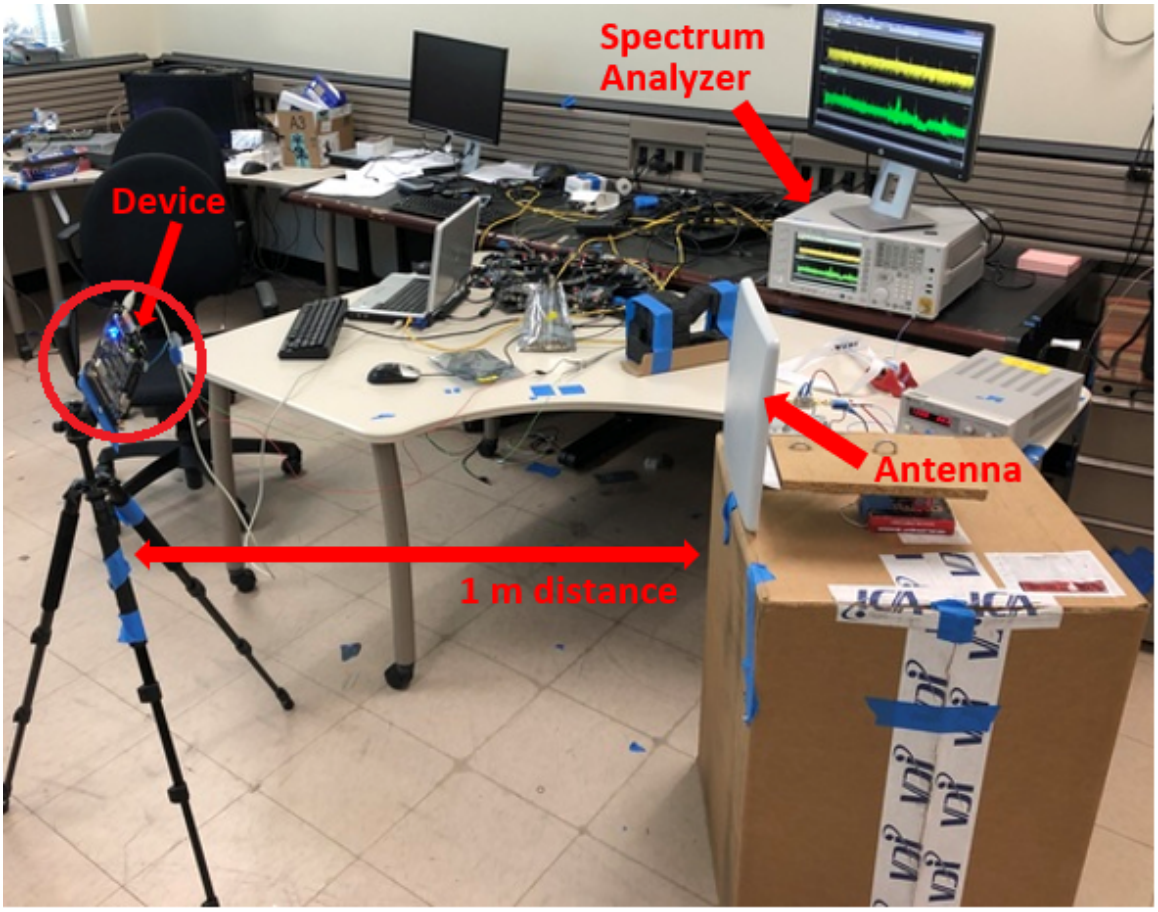


Figure 5.8: Experimental setup: monitoring from 1 m distance.

signal at 12.8 MHz sampling rate, and analyzed the digitized signal using P-TESLA.

Table 5.3: Mean accuracy for FPGA.

Benchmark	Path Prediction Accuracy
Print Tokens	98.7%
Replace	99.1%
Schedule	99.8%

Table 5.3 summarizes the mean accuracy. We observe that P-TESLA achieves excellent accuracy for monitoring all three benchmark applications, with roughly 99% accuracy for *Print Tokens* and *Replace*, and near-

perfect accuracy for *Schedule*.

Table 5.4: Mean timing difference for FPGA.

Benchmark	Mean Timing Difference
Print Tokens	0.98 samples
Replace	4.13 samples
Schedule	0.88 samples

We also report the mean timing difference of the predicted timestamps in Table 5.4. For *Print Tokens* and *Schedule* the mean timing difference is less than 1 sample. However, for *Replace*, the mean timing difference is roughly 4 samples. Note that, at the experimental sampling rate (12.8 MHz), 1 sample is equivalent to 78.125 ns. Thus, all timing estimations are very precise.

Monitoring from Distance: We further evaluate P-TESLA by monitoring the FPGA device from 1 m distance using a panel antenna. Figure 5.8 shows the experimental setup. We summarize the mean accuracy in Table 5.5. P-TESLA achieves better than 95% accuracy on all three benchmarks. In fact, for *Print Tokens* and *Schedule*, P-TESLA achieves roughly 99% accuracy.

Table 5.5: Mean accuracy at 1 m distance.

Benchmark	Path Prediction Accuracy
Print Tokens	98.68%
Replace	95.14%
Schedule	99.40%

Table 5.6: Mean timing difference at 1 m distance.

Benchmark	Mean Timing Difference
Print Tokens	3.78 samples
Replace	5.56 samples
Schedule	1.32 samples

Table 5.6 shows the mean timing difference for the predicted marker timestamps. While the timing differences are slightly higher than that of with probe, predicted timestamps are still very precise, and within a few samples.

While P-TESLA demonstrates excellent performance from 1 m distance, we notice slight degradation in accuracy compared to that of with probe (i.e., at 1 cm distance). This degradation is due to the lower SNR at distance, and can be improved by using high-gain antennas and/or low-noise amplifiers.

5.3.4 IoT Device Monitoring

We demonstrate the robustness of P-TESLA by monitoring an A13-OLinuXino IoT development board. This device has a 1 GHz Cortex A8 ARM processor [115]. Unlike the FPGA device, A13-OLinuXino runs on a Debian Linux operating system. We collected the EM side-channel signal by placing a magnetic probe near the microprocessor. The signal was recorded and demodulated using a spectrum analyzer (Agilent MXA N9020A). We then digitized the signal by passing it through an anti-aliasing filter with 20 MHz bandwidth, and sampling at 51.2 MHz sampling rate.

Table 5.7 shows the accuracy of P-TESLA for monitoring the IoT de-

Table 5.7: Mean accuracy for IoT device.

Benchmark	Path Prediction Accuracy
Print Tokens	94.15%
Replace	96.85%
Schedule	95.91%

vice. P-TESLA demonstrates high accuracy on all three benchmark applications; 94.15% on *Print Tokens*, 96.85% on *Replace*, and 95.91% on *Schedule*. Note that, P-TESLA achieves even higher accuracy (roughly 99%) for monitoring FPGA device. However, A13-OLinuXino has a much faster processor (1 GHz compared to FPGA's 50 MHz), which makes fine-grained execution monitoring more challenging. Furthermore, the operating system on A13-OLinuXino leads to more variations between training and testing executions. This, in turn, can cause performance degradation. As such, P-TESLA's performance on monitoring the IoT device is impressive.

Table 5.8: Mean timing difference for IoT device.

Benchmark	Mean Timing Difference
Print Tokens	7.68 samples
Replace	10.33 samples
Schedule	6.92 samples

Furthermore, the timing differences reported in Table 5.8 demonstrate that P-TESLA predicted timestamps are also quite precise. The mean timing difference for all three benchmark applications is within 10 samples. Note that, in our experiments (at 51.2 MHz sampling rate), each sample is equivalent to 19.5 ns.

Monitoring from Distance: We also evaluate P-TESLA by monitoring the IoT device from distance. For this, we placed a slot antenna at 1 m distance from the device.

Table 5.9: Mean accuracy at 1 m distance.

Benchmark	Path Prediction Accuracy
Print Tokens	89.87%
Replace	90.79%
Schedule	90.40%

Table 5.9 shows that P-TESLA achieves roughly 90% mean accuracy on all three benchmarks. Furthermore, Table 5.10 reports the mean timing difference. We also notice some performance degradation at distance. This is due to lower SNR that affects the signal matching adversely.

Table 5.10: Mean timing difference at 1 m distance.

Benchmark	Mean Timing Difference
Print Tokens	11.40 samples
Replace	33.66 samples
Schedule	7.53 samples

5.4 Summary

In this chapter, we presented P-TESLA, an approach for program execution tracing via EM side-channel signals. P-TESLA is completely non-invasive and does not impose any overhead on the monitored system. P-TESLA is especially useful for monitoring resource-constrained embedded devices

for tasks such as program debugging and anomalous/malicious program activity detection.

Experimental evaluations revealed that P-TESLA can provide highly accurate program traces for benchmark applications running on different embedded devices, even from 1 m away. Future research should focus on extending P-TESLA's capability by enhancing its signal processing techniques to monitor more sophisticated devices such as multi-core processors.

CHAPTER 6

IMPACTS OF SIGNAL QUALITY ON SIDE-CHANNEL ANALYSIS

6.1 Overview

Side-channels are often used in cryptanalysis to extract secret cryptographic keys, and more recently, for non-adversarial and non-intrusive program execution monitoring. Both these use cases have seen steady improvements, allowing ever-smaller differences in program behavior to be detected via side-channel analysis. However, it is still unclear where the limits for side-channel analysis are, e.g. whether a deviation of just a few instructions (or even a single instruction) in program execution can be identified from the side-channel signal, and how these limits depend on the rate at which the signal is observed (i.e., signal bandwidth), the amount of training that is available, and the quality of the signal (i.e., signal-to-noise ratio).

In this chapter, we investigate the practical limits of side-channel analysis. We demonstrate that given enough training data, sufficient signal bandwidth, high signal-to-noise ratio (SNR), and prior knowledge about the program code itself, even a single-instruction difference in the program execution can be identified with very high accuracy. For this, we monitor a popular open-source cryptographic software package (OpenSSL's sliding window modular exponentiation) via EM side-channel. Experimental evaluations demonstrate that each branch decision in the program execution can be recovered through the EM side-channel with high accuracy, using bandwidth and SNR that is easily provided by a sub-$1,000 equipment.

The main contributions of the chapter are:

- We demonstrate that branching decisions with even a single instruction deviation can be identified via EM side-channel analysis. Consequently, (1) detailed and accurate program execution tracking should be possible, and (2) secret-key-dependent branching should be avoided.

- We also show that large deviations in the program execution can be identified with >90% accuracy with a relatively narrow signal bandwidth (e.g., at a rate that is only 2% of the monitored system's clock cycle rate). However, identifying tiny deviations, such as a single instruction branching, requires a much faster signal rate (e.g., 8% or 16% of the monitored system's clock cycle rate). We further demonstrate that accurate program execution monitoring requires a relatively good signal quality (e.g., an SNR of 20 dB or higher).

The rest of this chapter is organized as follows. Section 6.2 describes the OpenSSL's sliding window modular exponentiation, Section 6.3 presents our method for side-channel-based branch decision prediction, Section 6.4 details experimental results, and Section 6.5 provides the summary.

6.2 Monitored Software - Modular Exponentiation in OpenSSL

We use an open-source implementation of the RSA public-key cryptosystem as a test subject to investigate the capacity/limitations of side-channels for identifying branching decisions in the program execution. Specifically, we try to recover the secret key bits from OpenSSL's sliding window modular exponentiation (function `BN_mod_exp_simple` from `bn_exp.c` in the OpenSSL source code) via EM side-channel analysis. This choice of software test subject has several key advantages. First, RSA's modular ex-

ponentiation has already been subjected to numerous side-channel attacks, so each branching decision (a decision to either continue to the next instruction in the program or jump to another instruction in the program) in its program code is extremely well understood. Second, many branching decisions depend on the bits of the secret exponent in RSA decryption, so our experimental results are directly relevant for cryptanalysis. Finally, the implementation already contains mitigation for previously demonstrated side-channel attacks, so the individual (secret-key-dependent) differences in program execution are very small and largely independent of each other, so our identification of each small difference in execution is not aided by signal changes caused by other correlated differences. The listing below shows the main part of this code, with some redaction (mainly removal of error checking) to enhance clarity:

```
1  wvalue = 0;                  /* The 'value' of the window */
2  wstart = bits - 1;           /* The top bit of the window */
3  wend = 0;                    /* The bottom bit of the window */
4  BN_one(r);
5  for (;;) {
6    if (BN_is_bit_set(p, wstart) == 0) {
7      BN_mod_mul(r, r, r, m, ctx);
8      if (wstart == 0)
9              break;
10     wstart--;
11     continue;
12   }
13   /* We now have wstart on a 'set' bit, we now need to work out
14   how big of a window to do. To do this we need to scan forward
```

```
15    until the last set bit before the end of the window */
16    j = wstart;
17    wvalue = 1;
18    wend = 0;
19    for (i = 1; i < window; i++) {
20      if (wstart - i < 0)
21        break;
22      if (BN_is_bit_set(p, wstart - i)) {
23        wvalue <<= (i - wend);
24        wvalue |= 1;
25        wend = i;
26      }
27    }
28
29    /* wend is the size of the current window */
30    j = wend + 1;
31    for (i = 0; i < j; i++)
32      BN_mod_mul(r, r, r, m, ctx);
33
34    /* wvalue will be an odd number < 2^window */
35    BN_mod_mul(r, r, val[wvalue >> 1], m, ctx);
36
37    /* move the 'window' down further */
38    wstart -= wend + 1;
39    wvalue = 0;
40    if (wstart < 0)
41      break;
42  }
```

This program code computes $v^p \ mod \ m$, where v is the (encrypted) message, p is the (secret) exponent, and m is the modulus. Conceptually, it follows the grade-school exponentiation approach, where the exponent is examined starting from the most significant bit, squaring the result for each bit of the exponent, and multiplying the result with the message when the bit of the exponent has a value of one. However, rather than process one bit of the exponent at a time, the sliding window algorithm splits the exponent into multi-bit chunks called *windows* and performs multiplication with the (appropriately pre-exponentiated) message an entire window at a time. In preparation for this, for each possible value of the window ($wvalue$), the value of $v^{wvalue} \ mod \ m$ has been pre-computed and placed in the table val at an index that corresponds to $wvalue$.

Initially, the very first window starts with a value of 0 (line 1), its most significant bit is the most significant bit of the exponent (line 2), and the window contains one bit from the exponent (line 3, note that the value of $wend$ is always one less than the number of bits in the window). Like the message and the exponent, the result r is a very large number (>1,000 bits) that is kept in a data structure that contains an array of 32-bit integers, so function BN_one sets the large number to a value of 1. The main loop of the exponentiation begins (line 5), and in each iteration of this a window is formed and the result is updated for that window. In this algorithm, a multi-bit window must begin and end with a non-zero bit. Therefore, line 6 examines the exponent's bit at the starting position of the window. If that bit is zero, that bit alone will be a (zero-valued) window. This means that the result should be squared (line 7) but no multiplication with the message is needed, so a new window will begin at the next bit position (line 10) and a new iteration of the main loop is begun (line 11). Note that,

if the bit that was just examined was the last (least significant) bit of the exponent (line 8), the entire exponentiation is now complete (line 9).

If the starting bit of the window is not zero, the code at lines 16 through 27 attempts to form a multi-bit window. Since the non-zero bit begins the window, the value of the window is now 1 (line 17), and the loop at line 19 iterates over the bits that are examined for potential inclusion into the window. Variable i is the number of bits that have been examined, and it starts at 1 because the very first (most significant) bit of the window has already been examined (and found to be 1). Line 20 checks if the would-be window would go past the last (least significant) bit of the exponent, in which case the window cannot be expanded further (line 21). Line 22 examines the next candidate bit. Since a window cannot end with a 0-valued bit, a 0-valued bit is simply skipped without expanding the window. However, a 1-valued bit causes the window to be expanded to include that bit, as well as all the zero-valued bits between it and the rest of the window. This is done by shifting the $wvalue$ by enough bit positions to make room for the bits that are being included (line 23), setting the least significant bit of $wvalue$ to 1 (line 24) because the new least significant bit of the window is 1 – note that the other bits that are being included into the window are all zero-valued, otherwise they would have been included when they were encountered – and setting $wend$ (line 25) to the new size of the window. The loop at line 19 ends when the number of bits examined for the current window reaches the maximum size allowed for a window. Variable $window$ stores this maximum size, and it is relatively small (5 or 6 for typical RSA key sizes) so that the table val (that contains a large number for every possible value of the window) is still small enough to fit in the processor's data cache (which is good for performance). Not every

window has the maximum size - the window's size depends on when a one-valued bit was last included. For example, if the maximum window size is 6 (i.e., $window = 6$), when the bits examined for the window are 100000, the window only has one bit (the initial 1), when the examined bits are 110000 the window has only two bits (11), when the examined bits 111000 or 101000 the resulting window has 3 bits (111 or 101), etc.

Once the window is formed, the result is squared for each bit in the window (lines 30-32) and then multiplied (line 35) with the value from table val that was pre-computed by exponentiating the message with the value of the window. Finally, $wstart$ and $wvalue$ are updated to begin a new window (lines 38 and 39), and another iteration of the main loop begins to form another window. The exception to this is when the just-processed window included the last bit of the exponent, in which case the entire exponentiation is complete (lines 40 and 41).

In prior cryptanalysis work, a cache-based attack [116, 117] was used to identify the sequence of squaring and multiplications, while for analog side channels the squaring and multiplications were identified in the signal by using especially chosen messages [69, 5] that create a large difference in their side-channel signals. Since each squaring corresponds to moving one bit toward the least significant bit of the exponent, and the multiplication corresponds to the end of a non-zero window, the sequence of squaring and multiplications reveals the position of each 1-valued bit that ends a window. Furthermore, when the number of squaring between two multiplications is less than the maximum allowed size of the window, that indicates the number of zero-valued bits that follow the last bit that was included in the window. By iteratively applying such exponent-reconstruction rules to the sequence of squaring and multiplications, a

significant percentage of the exponent's bits were recovered - 49% of the bits when the maximum window size is 4, and this percentage of recovered bits declines somewhat when the maximum window size is larger.

To avoid these attacks that rely on an exponent-dependent sequence of squaring and multiplications, OpenSSL has switched to a *fixed-window* exponentiation, where all windows are of the same size and all possible values of the window (including the all-zeros window) are represented in the pre-computed table val. This results in multiplications that are always separated by an equal number of squaring, regardless of the exponent. Furthermore, because every bit up to the window size is included in the window, there are no branch decisions that depend on the bits of the exponent - in fact, the sequence of *all* branch decisions is always the same for the entire exponentiation, regardless of the exponent. The listing below shows the main part of the exponentiation code (in function `BN_mod_exp_mont_consttime` within the `bn_exp.c` file in OpenSSL's source code), again with some redaction (mainly removing error-checking) for clarity:

```
1  while (bits >= 0) {
2    wvalue = 0;          /* The 'value' of the window */
3    /* Scan the window, squaring the result as we go */
4    for (i = 0; i < window; i++, bits--) {
5      BN_mod_mul_montgomery(&tmp, &tmp, &tmp, mont, ctx);
6      wvalue = (wvalue << 1) + BN_is_bit_set(p, bits);
7    }
8    /* Fetch the appropriate pre-computed value from the pre-buf */
9    MOD_EXP_CTIME_COPY_FROM_PREBUF(&am, top, powerbuf, wvalue,window);
10   /* Multiply the result into the intermediate result */
11   BN_mod_mul_montgomery(&tmp, &tmp, &am, mont, ctx);}
```

This code uses the more efficient Montgomery multiplication algorithm (`BN_mod_mul_montgomery`) that improves resilience to cache-based side-channel attacks using a more complicated organization of the lookup table (the variable that refers to it is named $powerbuf$ in this code, rather than val) that requires a specialized function `MOD_EXP_CTIME_COPY_FROM_PREBUF` to retrieve the table entry that corresponds to $wvalue$ and place it into am so it can be multiplied with the result.

Because the existing branch decisions in this code leak no information about the exponent, we change the code so that line 6 is replaced by

```
1       wvalue = (wvalue << 1)
2       if(BN_is_bit_set(p, bits))
3         wvalue|=1;
```

Note that this introduces an exponent-dependent branch decision which creates a single-instruction difference in program execution, so it will allow us to experimentally assess how accurately a single-instruction difference in execution can be identified from the side-channel signal.

6.3 Side-Channel-Based Branch Decision Prediction

The branching decision prediction method consists of two phases: the training phase and the prediction phase. In the training phase, we learn the EM signatures corresponding to each branch decision by executing encryption with known keys. In the prediction phase, we use the learned EM signatures from training to predict which branch decision is the best match for the EM signal observed during prediction. In both cases, we need to capture and pre-process the signals before running our prediction algorithm.

6.3.1 Acquisition of Modulated EM Signals

Unintentional EM emanations occur at various frequencies, but of particular importance is the frequency band centered around the clock frequency of the device's processor and memory. This frequency band contains signals that are primarily a function of the instruction sequence executed by the CPU. Each processor cycle, the CPU draws a current that is a direct result of the executed instruction(s). Much of this instruction-dependent current is drawn by the CPU clock circuitry and by the circuitry that performs new computations (i.e., switches on and off) every CPU clock cycle. This creates a strong current at the CPU clock frequency, which acts as a carrier modulated by the clock-cycle-to-clock-cycle variations in program activity (i.e., executed instructions). When its EM signal is observed this way, the computer system has much in common with a communications system: the processor is a transmitter which (inefficiently and unintentionally) transmits a carrier (i.e., the clock signal) that is amplitude-modulated by a program's activity as it causes activity in the processor's circuitry. We can then receive and demodulate this signal using wireless communications techniques. All EM signals, both in the training phase and in the monitoring phase, are first AM demodulated at the processor clock frequency, low-pass filtered and sampled before being sent for signal processing.

6.3.2 Signal Processing

The first step in both training and testing is to identify the location where either control-flow branching or window value calculation takes place in the received signal. Since the modular multiplication of large numbers (functions `BN_mod_mul` and `BN_mod_mul_montgomery`) has a fixed sequence

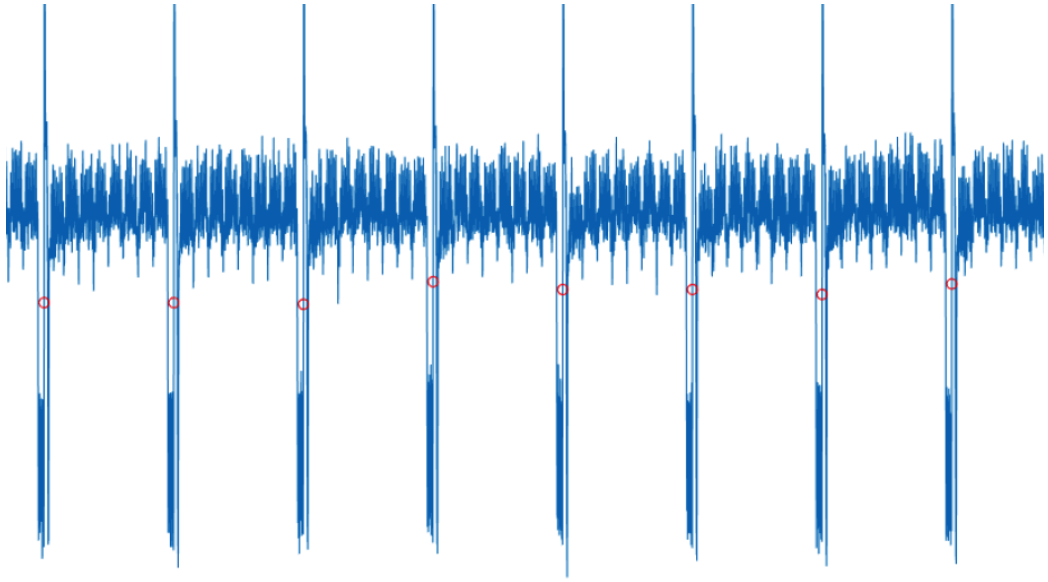


Figure 6.1: The EM signature of $BN\_mod\_mul\_montgomery$ function.

of branch decisions and is thus of no interest for our experiments. We find the signature of this multiplication and identify the part of its signal that has a prominent and abrupt change in the signal (see Figure 6.1). Because modular multiplication was designed to have almost no timing variation, the end of the signal that corresponds to it can be found at a fixed time offset from the point of this match. The signal that we use for training and prediction of exponent-dependent branches consists of the snippets of the signal between each ending of a modular multiplication and the beginning of the next one.

6.3.3 Training Phase

In the training phase, we execute encryption with known keys, select the snippets of the signal between modular multiplications, and use them to create a “dictionary" of reference EM patterns for each possible combination of branch outcomes within the code the snippet corresponds to. We first label each snippet according to which modular multiplications it has

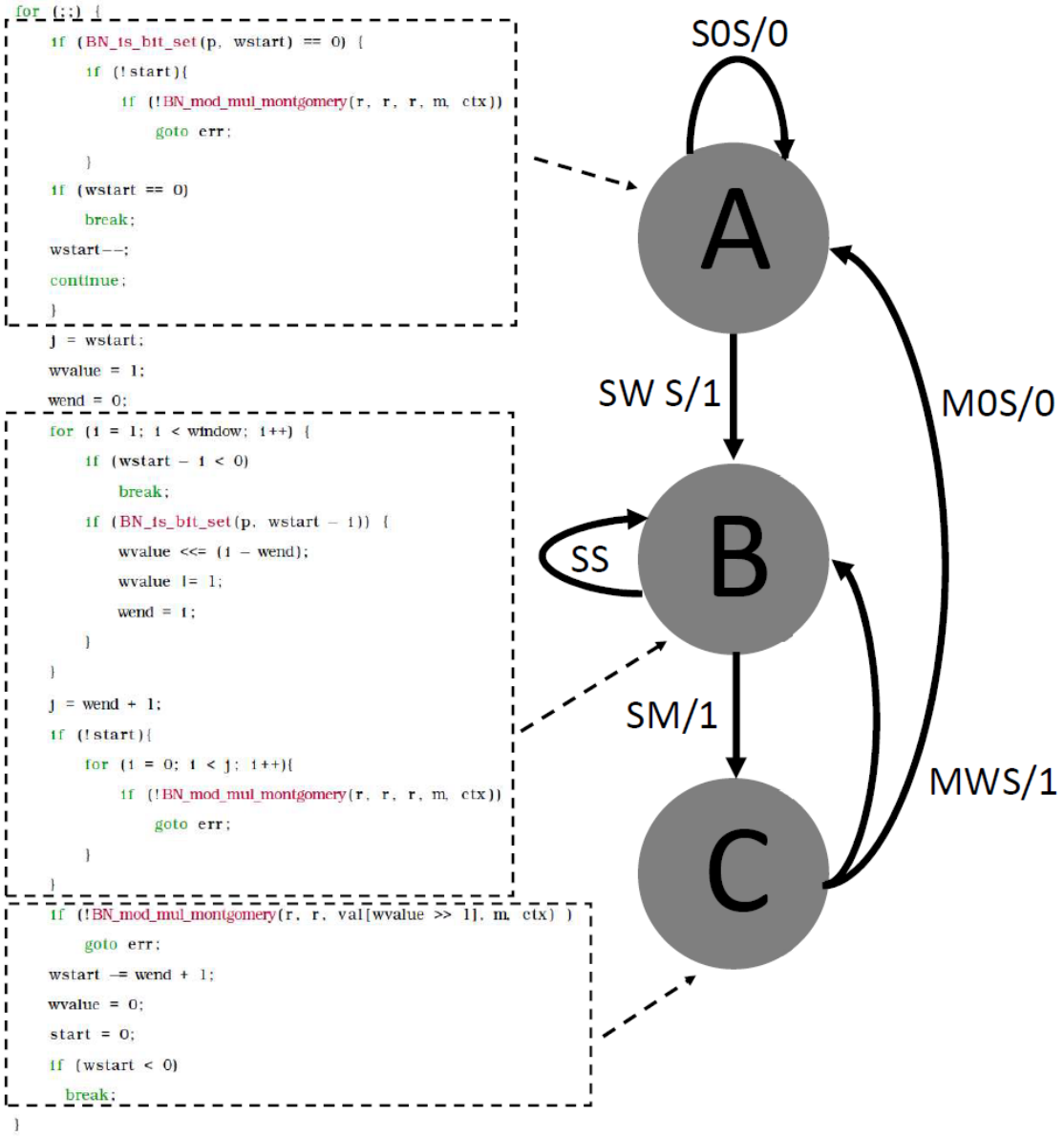


Figure 6.2: Labeling of the signal snippets for sliding-window exponentiation.

at its beginning and its end and treat these snippets as transitions in a state machine whose states are the modular multiplications as shown in Figure 6.2. In addition to this, the transition from state A to state B (which corresponds to examining the candidate bits for a non-zero window) has 32 different sub-labels that correspond to all possible combinations of outcomes for the five branch outcomes that examine on the candidate bits.

Since training is performed with known exponents, the labels for all the snippets collected during training are also known, so the output of the training is a "dictionary" that contains a large number of labeled signal snippets.

6.3.4 Prediction Phase

During the detection phase, just like in training, we first identify and extract the snippets of the signal that correspond to execution between modular multiplications. Then, for each snippet encountered in the detection phase, we use the 1-Nearest Neighbor (1-NN) algorithm, with Euclidean distance as the distance metric, to find which snippet in the dictionary is the closest match for that detection-phase snippet, and use the label of that dictionary snippet to label the detection-time snippet.

6.4 Experimental Evaluations

6.4.1 Experimental Setup

We run the OpenSSL's RSA decryption on an embedded device (A13 OLinuXino board [115]). The A13- OLinuXino board is a single-board computer that has an in-order 2-issue Cortex A8 ARM processor [118], and it uses the Debian Linux operating system.

Signals are received using a small magnetic probe. We place the probe close to the monitored system as shown in Figure 6.3. The signals collected by the probe are recorded with Keysight N9020A MXA spectrum analyzer [119]. Our decision to use a spectrum analyzer was mainly driven by its existing features such as built-in support for automating measurements, saving and analyzing measured results, visualizing the signals when debugging code, etc. We have observed very similar signals when

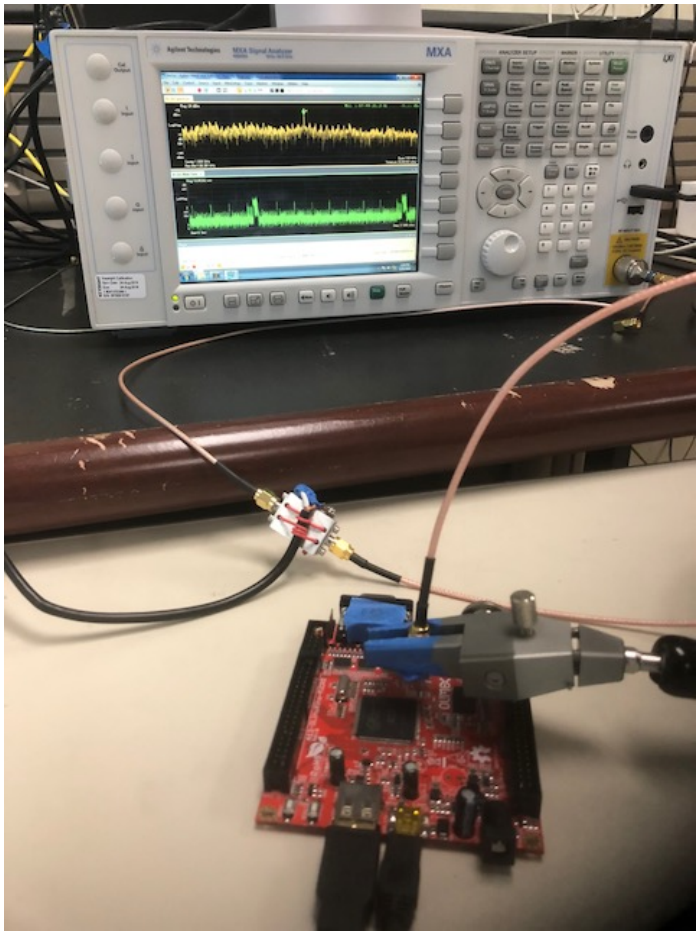


Figure 6.3: Experimental setup.

using less expensive equipment such as Ettus USRP B200-mini receiver [95]. The analysis was implemented in MATLAB and executed on a personal computer, takes less than one minute to reconstruct the exponent-dependent branch decisions encountered in one 1024-bit modular exponentiation.

6.4.2 Impact of Signal Bandwidth on Branch Decision Reconstruction

The branch decisions we are interested in are the exponent-dependent branches in the sliding window implementation and in the modified fixed window implementation. These can be clustered into three categories:

First, branches at lines 8 and 37 in the sliding-window implementa-

tion result in relatively large changes in what is executed after them. The branch at line 8 results in either calling `BN_mod_mul` at line 9, or in executing the entire window-forming loop (lines 22-33), followed by lines 36 and 37 before `BN_mod_mul` is entered at line 38. The loop branch at line 37 results in either calling `BN_mod_mul` at line 38, or in exiting the loop, calling `BN_mod_mul` at line 41, and then executing the code at lines 43-47 and entering another iteration of the main exponentiation loop. Compared to the branch at line 8, the outcome of this branch is more difficult to identify using the side channel signal because the long-lasting `BN_mod_mul` is called almost immediately after this branch, regardless of its outcome, and the largest difference that results from its outcome occurs only *after* `BN_mod_mul` returns.

Second, the branch at line 28 in the sliding window implementation results in either executing the code at lines 29-31, which consists of only four processor instructions, or not executing these four instructions. After that, the two options converge.

Finally, the exponent-dependent branch in our modified fixed-window implementation creates only a single-instruction (a bitwise OR instruction) difference in what is executed by the processor.

Figure 6.4 shows the accuracy of identifying (reconstructing) branch decisions for these three kinds of branches, when the signal is received using 20 MHz, 30 MHz, 40 MHz, 80 MHz, and 160 MHz of bandwidth. Note that the processor clock frequency is 1 GHz, so these bandwidth values correspond to only 2%, 3%, 4%, 8%, and 16% of the processor's clock frequency. From these results, we can see that branch decisions which create large changes in subsequent program execution can be reconstructed nearly perfectly even when using limited signal bandwidth. For branches

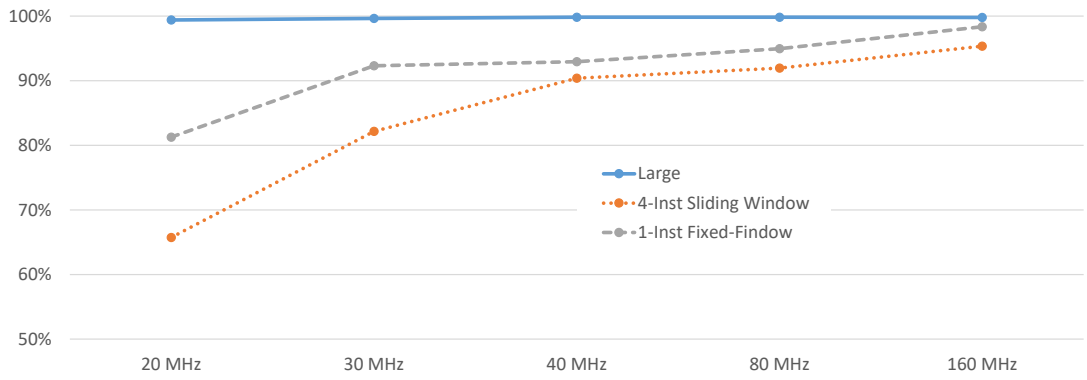


Figure 6.4: Accuracy of side-channel-based reconstruction of branch decisions (vertical axis) for different values of the received signal bandwidth (horizontal axis).

that cause 4-instructions and single-instruction changes in program execution, some reconstruction is possible even when using bandwidth that is only 2% of the processor's clock frequency, but reconstruction accuracy significantly improves when bandwidth is increased to 4% of the clock frequency, and then modestly improves as bandwidth is expanded to 8% and then 16% of the processor's clock frequency. However, we found it surprising that reconstructing accuracy for 1-instruction differences in our modified fixed-window implementation was consistently better than the reconstruction accuracy for 4-instruction differences in the sliding-window implementation. Upon further investigation, we have found that, in addition to signal bandwidth and the amount of execution change, the accuracy of branch decision reconstruction also depends on how much is known about the branches that belong to the same signal "snippet" during analysis. In the fixed-window implementation, the outcomes of all branch decisions in the signal "snippet" are already known (because the entire exponentiation follows a fixed sequence of branch decisions), so only one bit of information is extracted from the signal "snippet", i.e. the reconstruction decision

must choose between only two possibilities. In contrast, in the sliding-window implementation, five exponent-dependent branches occur in the same signal "snippet", so five bits of information must be extracted from the signal "snippet", i.e. the reconstruction decision is a choice among 32 possibilities.

Our results for reconstruction of branch decisions that cause large changes in program execution are an improvement over the prior state of the art in analog side-channel cryptanalysis [69, 5] in at least two significant ways. That prior work [69, 5] relies on specially crafted messages (a chosen-cyphertext attack) that cause signals for a squaring (result-result multiplication) and a (result-message) multiplication to differ significantly, which allows the sequence of squaring and multiplication operations to be recognized in the sliding-window implementation, and then some bits of the key can be recovered from the squaring-multiplication sequence. The first advantage of our approach, when used for cryptanalysis, is that it works for any message, and in fact, requires no knowledge of the message at all. The second cryptanalysis advantage of our approach is that, instead of distinguishing between squaring and multiplication operations, it reconstructs the exponent-dependent branch decisions (branches at lines 8 and 37 in the sliding-window code) that decide not only whether a squaring or multiplication should be executed next, but also *which* squaring operation follows - the one used for a single-bit zero-valued window (line 9) or the one for a non-zero window (line 38). This allows recovery of more of the exponent's bits – we discover at which bit-position each non-zero window begins and ends, and also at which bit positions all the single-bit zero-valued windows are, so the only bits left unknown are those between the leading 1 and the trailing 1 in each non-zero window. The effect of this is

that where prior work was able to recover 49% of the exponent's bits when the maximum window size was 4 (and fewer bits when the maximum window size is larger), in our experiments the maximum window size was 6 and yet 55% of the exponent's bits are recovered using reconstruction of branch decision of only the large-change branches (those at lines 8 and 37 in the sliding-window program listing).

Furthermore, successful reconstruction of branch decisions that cause 4-instruction changes in program execution would recover *all* of the exponent's bits. Our experiments show that, when the received signal's bandwidth is 4% or more of the clock frequency, these 4-instruction-change branch decisions are recovered with >90% accuracy, and that accuracy exceeds 95% when using 16% bandwidth, and this alone allows recovery of 90% to 95% of the exponent's bits, far more than in prior work. Finally, note that information gained from the reconstruction of large-change and 4-instruction-change decisions can be combined, leading to the recovery of 84% of the exponent's bits even with 2% bandwidth, 95% with 4% bandwidth, and 98% with 16% bandwidth. For side-channel-based program monitoring, this implies that knowledge of how the branch decisions in the program are related to each other can be used to improve the overall accuracy of tracking the program at basic-block granularity.

Finally, our results for 1-instruction-change branches in the modified fixed-window implementation further imply that *any* key-dependent branching should be avoided in cryptographic implementations, and that side-channel-based program monitoring with single-instruction granularity is *possible*, for at least some parts of the code.

6.4.3 Impact of Training on Branch Decision Reconstruction

Another important factor in both cryptanalysis and side-channel based execution monitoring is how much training is needed to achieve a desired level of accuracy, and how the accuracy improves as more or less training is available. To help answer this question, we focus on the branch that causes a 4-instruction change in the sliding-window exponentiation (the branch at line 28). The results of this study are shown in Figure 6.5, for signals received with a bandwidth of 160 MHz (16% of the processor's clock frequency). The smallest amount of training we use consists of only one exponentiation, but since 2048-bit RSA decryption uses two 1024-bit exponents, even a single (1024-bit) exponent results in 130 to 140 signal "snippets" that correspond to window-forming loop (lines 25-33 in the sliding-window listing). Recall that for each such signal "snippet" the reconstruction decision has $2^5 = 32$ possible outcomes because each snippet contains 5 execution instances of the branch at line 28, so each 1024-bit exponentiation used in training on average provides slightly more than 4 signal snippets for each of these possibilities.

We observe that training with only 4 examples for each distinct 5-branch reconstruction possibility results in 62% reconstruction accuracy, which is not very accurate but is noticeably more than the 50% accuracy that would result from a purely random reconstruction. The accuracy dramatically improves with more training, reaching 86% when 85 training examples for each 5-branch reconstruction possibility, 95% when 1028 training examples per reconstruction possibility are available, and the improvement in accuracy continues, albeit more slowly, as even more training examples are added.

For cryptanalysis, this amount of training is not difficult to obtain -

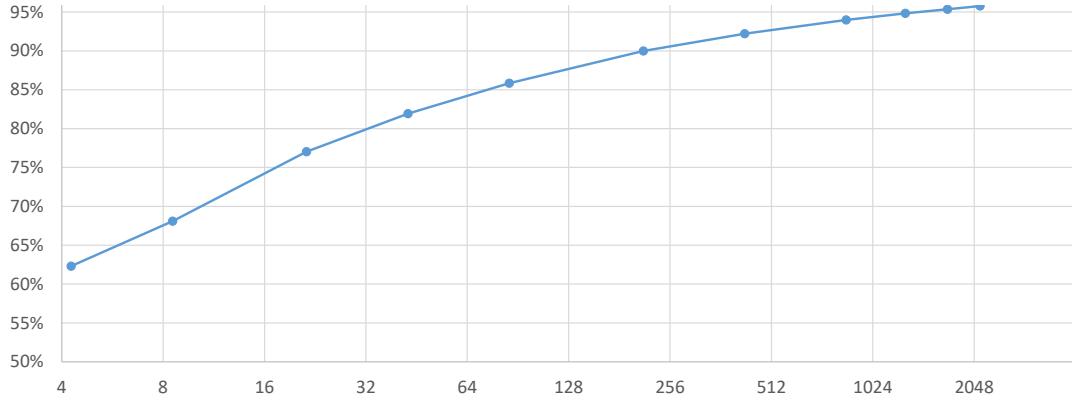


Figure 6.5: Accuracy of side-channel-based reconstruction of branch decisions (vertical axis) when changing the number of training examples for that point in the program code (horizontal axis, note the logarithmic scale).

even the rightmost data point in Figure 6.5 corresponds to using only 250 RSA decryptions (with known keys) for training, and this training can be accomplished in only a few minutes and the entire "dictionary" produced by training occupies only a few tens of megabytes of memory. For side-channel-based program monitoring, however, these results imply that training may possibly need to contain hundreds of signal examples for each point in the code, which is likely to create two kinds of problems. First, in any given application, some parts of the program are very rarely executed, so it may be hard to rapidly obtain enough signal examples for those parts of the code. Second, for very large software, hundreds of signal examples for each fine-grained part of the code, with possibly millions of such points in the code, would require an enormous memory/disk footprint to store all of the training data that is needed to monitor such an application. This means that significant further research may be needed on how to summarize and/or compress the training data without sacrificing reconstruction accuracy.

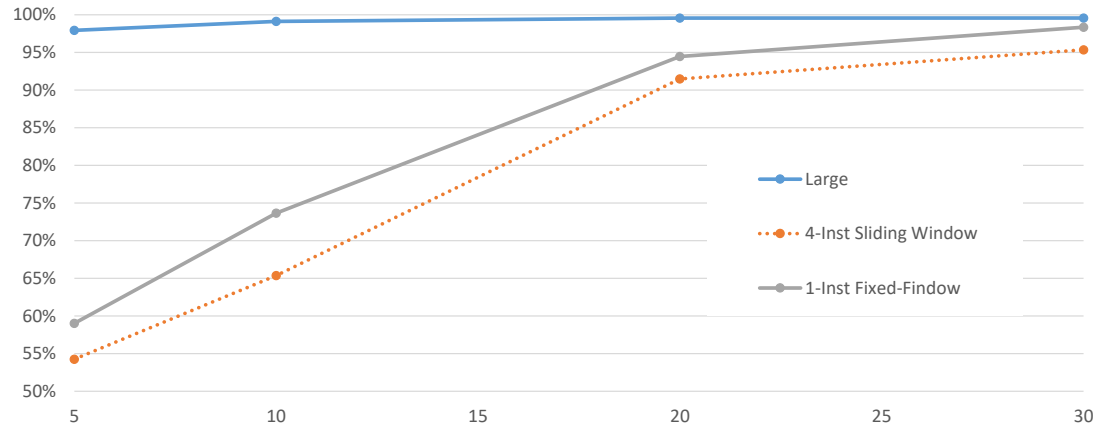


Figure 6.6: Accuracy of side-channel-based reconstruction of branch decisions (vertical axis) when changing the SNR (horizontal axis in decibels (dB), which implies a logarithmic scale).

6.4.4 Impact of SNR on Branch Decision Reconstruction

Our results shown thus far use the signal collected in very close proximity (about 2 cm) of the monitored system, where the Signal-to-Noise Ratio (SNR) is about 30 dB, i.e. the power of the received signal is about 1,000 times the power of the received noise. To study how the SNR of the side channel signal affects the reconstruction of branch decisions, we perform additional experiments where we reduce the SNR to 20 dB (signal has 100 times the power of the noise), 10 dB (signal has 10 times the power of the noise), and 5 dB (signal has only 3.2 times the power of the noise). We again use 160 MHz of bandwidth, and in Figure 6.6 show results for each of the three categories of branches – large-change and 4-instruction-change branches from the sliding-window exponentiation, and the 1-instruction-change branches from our modified fixed-window implementation.

From these results, we observe that a relatively low SNR (5 dB) still allows highly accurate (98%) reconstruction of branch decisions that cause

large changes in program execution, while the reconstruction of small-change branch decisions is only somewhat better (than pure random guessing (50%)). Increasing the SNR to 10 dB and then to 20 dB brings the large-change branch decisions very close to perfect reconstruction accuracy, and also dramatically improves reconstruction for small-change branch decisions, which at 20 dB SNR can be reconstructed with >90% accuracy. Finally, increasing the SNR to 30 dB has little impact on the (already close-to-perfect) reconstruction accuracy for large-change branch decisions, but does improve reconstruction accuracy for small-change branch decisions, which are not reconstructed with >95% accuracy.

For cryptographic implementations, these results imply that branch decisions that lead to large changes in what is subsequently executed would not be based on key material should, because that allows key material to be recovered through side-channel signals even when they are received with low SNR, i.e., from larger distances or in the presence of some noise-generating and/or shielding countermeasures. Accurate reconstruction of branch decisions that result in changing the program execution by only one or a few instructions, however, does require a reasonably good SNR (at least 20 dB), so countermeasures that dramatically reduce the SNR of the signal may be effective.

For side-channel-based monitoring of program execution, these results imply that significant changes in program execution, e.g., tracking which coarse-grained part of the program is executed or detecting long bursts of anomalous execution, can use signals received with relatively low SNR, but fine-grained (instruction-level) tracking of program execution and/or detection of small divergences from normal execution requires good-quality (at least 20 dB SNR).

6.5 Summary

In this chapter, we investigate how the decisions of branches (i.e., the control-flow decisions) in the program execution can be recovered (reconstructed) by monitoring side-channel signals. For this, we use a popular open-source cryptographic software (RSA OpenSSL) as a test subject and evaluate how the accuracy of the control-flow reconstruction depends on the signal's received bandwidth, the amount of training data, and the signal-to-noise ratio. Our results indicate that, with reasonable signal bandwidth, enough training, and a good signal-to-noise ratio, even single instruction differences in the program control flow can be recovered from the electromagnetic (EM) emanations. We also observe that branch decisions that result in coarse-grained changes in the program execution can be reconstructed even using signals that have relatively low bandwidth and SNR. Consequently, to protect against side-channel attacks, branch decisions that consider fine-grained parts (e.g., individual bits) of the cryptographic key should be avoided, especially when these branch decisions result in coarse-grained changes in the subsequent execution. Our results also imply that side-channel-based non-adversarial monitoring of the program execution (e.g., to obtain runtime performance information or detect anomalies) is possible using signals collected with reasonable and compact equipment, e.g., sub-$1,000 receivers that provide >40 MHz of bandwidth and compact probes placed a few centimeters from the monitored system. However, monitoring of large applications requires enough training for the rarely-executed parts of the application.

CHAPTER 7

RESEARCH CONTRIBUTIONS AND FUTURE WORK

7.1 Research Contributions

This research proposed to leverage side-channels for non-adversarial and non-intrusive monitoring of embedded and cyber-physical systems. We demonstrated that EM side-channel signal analysis can be an effective way for protecting resource-constrained security-critical embedded devices from stealthy intrusions and malicious attacks. We also designed a framework for detailed (basic-block granularity) program execution tracing exploiting the device's EM side-channel signal. Furthermore, we showed that given enough training data, wide monitoring bandwidth, and high signal-to-noise ratio, even a single instruction deviation in the program execution can be detected with high accuracy through EM side-channel analysis. The techniques proposed in this thesis provide solutions for identifying deviations in the device's EM side-channel signals for anomaly-based intrusion/malware detection and for establishing a correspondence between training and testing signals for detailed program execution tracing. The main research contributions of this thesis are:

1. We have designed an anomaly-based intrusion detection system called IDEA (Intrusion Detection through Electromagnetic signal Analysis) [48]. In the training phase, IDEA creates a dictionary of trusted EM signatures by monitoring a reference device. IDEA is next deployed to monitor a target device. IDEA monitors and compares the device's EM emanations to the reference EM signatures, and reports intru-

sion when deviations in EM side-channel signal are detected. IDEA can effectively monitor and protect different embedded devices (FPGAs, IoTs, CPSs, etc.) against stealthy intrusions with high detection accuracy (AUC $\approx$ 0.99) from distances up to 3 m.

2. We have designed a neural network to model the device's EM side-channel signal and detect malicious activities in embedded devices through identifying deviations in EM emanations [49]. The malware detection system trains with EM signals from an uncompromised or 'malware-free' reference device to model normal program activity. We then exploit this trained model to monitor EM emanations from target devices. When the device performs anomalous/malicious activity, the device's EM side-channel signal violates the learned signal model. This, in turn, increases the neural network's prediction error, which we detect and report as anomalous/malicious activity. The system can detect even stealthy (e.g., 5 μs long) malicious attacks with high accuracy (AUC $\approx$ 0.99) and low detection latency ($\approx$ 20 μs) from distances up to 3 m. Note that, while its detection performance is comparable to IDEA [48], the system is roughly 35 times faster and 375 times memory efficient than that of the IDEA framework.

3. We have designed a novel framework called P-TESLA (Program-Tracing through Electromagnetic Side-channeL Analysis) [50] that leverages the device's EM side-channel signals for zero-overhead and non-intrusive program execution tracing. P-TESLA has a two-step training process in which the instrumented training facilitates to annotate the uninstrumented training signals and establish a correspondence between code segments (i.e., basic blocks) and signal snippets. Furthermore,

P-TESLA exploits a novel signal matching technique that efficiently matches the test signal with the training signals to reconstruct the program execution path. P-TESLA can effectively monitor different embedded devices (e.g., FPGAs and IoTs) and reconstruct detailed (basic-block granularity) program execution traces with high ($\approx$ 99%) accuracy from up to 1 m distance.

4. We have demonstrated that even a single instruction deviation in the program execution can be detected with high accuracy via EM side-channel signal analysis. However, to successfully monitor fine-grained program activity, the monitoring system requires sufficient signal bandwidth (e.g., roughly 8% to 16% of the monitored system's clock speed), enough training data (e.g., nearly 100% code-coverage), adequate signal-to-noise ratio (e.g., 20 dB or higher), and prior knowledge about the program code and its vulnerabilities. Note that such monitoring bandwidth and SNR can be achieved by a relatively cheap (e.g., a sub-$1,000) equipment. Thus, we have concluded that EM side-channel based non-adversarial program execution monitoring is very much feasible. We have also demonstrated that branching decisions in the program execution are easily identifiable in side-channel attacks, and thus, any secret-key-dependent branching decision should be avoided.

7.2 Future Research Directions

Although this thesis provides techniques that leverage side-channels to monitor embedded devices, our techniques, in their current settings, do not monitor multiple devices simultaneously. Thus, future research should focus on extending these techniques to monitor multiple devices using a

single monitoring system (i.e., with a single antenna and a single receiver). This would significantly reduce the deployment cost, making the monitoring system more suitable for commercial deployment.

Another interesting direction for research is cross-device side-channel analysis. Existing side-channel analysis techniques require device-specific training (i.e., separate and unrelated training for each device-class). However, researchers have demonstrated that different devices behave similarly (and emanate similar side-channel signals) while executing similar program activities. For instance, periodic activities, such as program loops, cause periodic signals (in both EM and power side-channels) with loop-specific spectral spikes, regardless of the device-class. Similarly, in all devices, the last-level cache misses in program execution cause dips in the monitored side-channel signal amplitude. This indicates that cross-device side-channel analysis is feasible and training from one device-class can be exploited for analyzing a different device-class. Therefore, developing a cross-device side-channel analysis platform could reduce the training time or the data collection time significantly.

REFERENCES

[1] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973.

[2] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*, Springer, 1999, pp. 388–397.

[3] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2015, pp. 207–228.

[4] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. Callan, A. Zajic, and M. Prvulovic, "One&done: A single-decryption em-based attack on openssl's constant-time blinded rsa," in *Proceedings of the 27th USENIX Conference on Security Symposium*, USENIX Association, 2018, pp. 585–602.

[5] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *International Cryptology Conference*, Springer, 2014, pp. 444–461.

[6] C. R. A. González and J. H. Reed, "Power fingerprinting in sdr integrity assessment for security and regulatory compliance," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, p. 307, 2011.

[7] O. Söll, T. Korak, M. Muehlberghuber, and M. Hutter, "Em-based detection of hardware trojans on fpgas," in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, IEEE, 2014, pp. 84–87.

[8] J. Balasch, B. Gierlichs, and I. Verbauwhede, "Electromagnetic circuit fingerprints for hardware trojan detection," in *2015 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, Aug. 2015, pp. 246–251.

[9] R. Richards, "High-assurance cyber military systems (hacms)," *DARPA. mil*, 2016.

[10] E. Colbert, "Security of cyber-physical systems," *Journal of Cyber Security and Information Systems*, vol. 5, no. 1, 2017.

[11] G. D. Maayan, "The iot rundown for 2020: Stats, risks, and solutions," *Security Today*, Jan. 2020, https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx?Page=2.

[12] *INTEL a guide to the internet of things infographic*, https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html, Accessed: 2018-03-01.

[13] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, IEEE, 2015, pp. 145–152.

[14] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[15] E. Nakashima and S. Mufson, "Hackers have attacked foreign utilities, cia analyst says," *Washington Post*, Jan. 2008.

[16] J. Sametinger, J. Rozenblit, R. Lysecky, and P. Ott, "Security challenges for medical devices," *Communications of the ACM*, vol. 58, no. 4, pp. 74–82, 2015.

[17] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Ddos in the iot: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[18] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security—a survey," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1802–1831, 2017.

[19] K. Dunham, "Evaluating anti-virus software: Which is best?" *Information Systems Security*, vol. 12, no. 3, pp. 17–28, 2003.

[20] A. Mohaisen and O. Alrawi, "Av-meter: An evaluation of antivirus scans and labels," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2014, pp. 112–131.

[21] C Foundation, *Automated malware analysis - cuckoo sandbox*, http://www.cuckoosandbox.org/.

[22] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.

[23] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 289–302, 2016.

[24] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 41, 2013, pp. 559–570.

[25] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, 2016.

[26] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2014, pp. 109–129.

[27] A. Viswanathan, K. Tan, and C. Neuman, "Deconstructing the assessment of anomaly-based intrusion detectors," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2013, pp. 286–306.

[28] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, IEEE, 1999, pp. 133–145.

[29] B. B. Kang and A. Srivastava, "Dynamic malware analysis," in *Encyclopedia of Cryptography and Security, 2nd Ed.* 2011, pp. 367–368.

[30] N. Scaife, H. Carter, P. Traynor, and K. R. Butler, "Cryptolock (and drop it): Stopping ransomware attacks on user data," in *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, IEEE, 2016, pp. 303–312.

[31] K. Tian, D. Yao, B. G. Ryder, and G. Tan, "Analysis of code heterogeneity for high-precision classification of repackaged malware,"

in *Security and Privacy Workshops (SPW), 2016 IEEE*, IEEE, 2016, pp. 262–271.

[32] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices.," in *HealthTech*, 2013.

[33] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "Eddie: Em-based detection of deviations in program execution," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, IEEE, 2017, pp. 333–346.

[34] Y. Han, S. Etigowni, H. Liu, S. Zonouz, and A. Petropulu, "Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 1095–1108.

[35] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso, "Zero-overhead profiling via em emanations," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016, pp. 401–412.

[36] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, IEEE, 2016, pp. 1–11.

[37] Y. Liu, L. Wei, Z. Zhou, K. Zhang, W. Xu, and Q. Xu, "On code execution tracking via power side-channel," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 1019–1031, ISBN: 978-1-4503-4139-4.

[38] G. A. Jacoby, R. Marchany, and N. Davis, "Battery-based intrusion detection a first line of defense," in *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, IEEE, 2004, pp. 272–279.

[39] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proceedings of the 6th international conference on Mobile systems, applications, and services*, ACM, 2008, pp. 239–252.

[40] L. Liu, G. Yan, X. Zhang, and S. Chen, "Virusmeter: Preventing your cellphone from spies," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2009, pp. 244–264.

[41] T. K. Buennemeyer, T. M. Nelson, L. M. Clagett, J. P. Dunning, R. C. Marchany, and J. G. Tront, "Mobile device profiling and intrusion detection using smart batteries," in *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, IEEE, 2008, pp. 296–296.

[42] J. R. Larus, "Efficient program tracing," *Computer*, vol. 26, no. 5, pp. 52–61, 1993.

[43] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, IEEE, 2002, pp. 291–301.

[44] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *Proceedings of the 2nd international conference on Virtual execution environments*, 2006, pp. 154–163.

[45] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, pp. 1319–1360, 1994.

[46] B. B. Yilmaz, E. M. Ugurlu, F. Werner, M. Prvulovic, and A. Zajic, "Program profiling based on markov models and em emanations," in *Cyber Sensing 2020*, International Society for Optics and Photonics, vol. 11417, 2020, p. 114170D.

[47] B. B. Yilamz, E. M. Ugurlu, A. Zajic, and M. Prvulovic, "Instruction level program tracking using electromagnetic emanations," in *Cyber Sensing 2019*, International Society for Optics and Photonics, vol. 11011, 2019, 110110H.

[48] H. A. Khan, N. Sehatbakhsh, L. N. Nguyen, R. L. Callan, A. Yeredor, M. Prvulovic, and A. Zajic, "Idea: Intrusion detection through electromagnetic-signal analysis for critical embedded and cyber-physical systems," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[49] H. A. Khan, N. Sehatbakhsh, L. N. Nguyen, M. Prvulovic, and A. Zajić, "Malware detection in embedded systems using neural network model for electromagnetic side-channel signals," *Journal of Hardware and Systems Security*, vol. 3, no. 4, pp. 305–318, 2019.

[50] H. A. Khan, S. Park, A. Zajić, and M. Prvulovic, "P-tesla: Program-tracing through electromagnetic side-channel analysis," *IEEE Transactions on Computers (submitted)*, 2020.

[51] H. A. Khan, M. Alam, A. Zajic, and M. Prvulovic, "Detailed tracking of program control flow using analog side-channel signals: A promise for iot malware detection and a threat for many cryptographic implementations," in *Cyber Sensing 2018*, International Society for Optics and Photonics, vol. 10630, 2018, p. 1 063 005.

[52] W. H. Ware, "Security and privacy in computer systems," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM, 1967, pp. 279–282.

[53] W. Van Eck, "Electromagnetic radiation from video display units: An eavesdropping risk?" *Computers & Security*, vol. 4, no. 4, pp. 269–286, 1985.

[54] H. J. Highland, "Electromagnetic radiation revisited," *Computers & Security*, vol. 5, no. 2, pp. 85–93, 1986.

[55] P Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proceedings of CRYPTO'96, Springer, Lecture notes in computer science*, 1996, pp. 104–113.

[56] W Schindler, "A timing attack against RSA with Chinese remainder theorem," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000*, 2000, pp. 109–124.

[57] D. Boneh and D. Brumley, "Remote Timing Attacks are Practical," in *Proceedings of the USENIX Security Symposium*, 2003.

[58] L Goubin and J Patarin, "DES and Differential power analysis (the "duplication" method)," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999*, 1999, pp. 158–172.

[59] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Power analysis attacks of modular exponentiation in smart cards," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999*, 1999, pp. 144–157.

[60] S Chari, C. S. Jutla, J. R. Rao, and P Rohatgi, "Towards sound countermeasures to counteract power-analysis attacks," in *Proceedings of CRYPTO'99, Springer, Lecture Notes in computer science*, 1999, pp. 398–412.

[61] A. G. Bayrak, F Regazzoni, P Brisk, F.-X. Standaert, and P Ienne, "A first step towards automatic application of power analysis countermeasures," in *Proceedings of the 48th Design Automation Conference (DAC)*, 2011.

[62] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "Ecdsa key extraction from mobile devices via nonintrusive physical side channels," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1626–1638.

[63] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer, "Physical key extraction attacks on pcs," *Communications of the ACM*, vol. 59, no. 6, pp. 70–79, 2016.

[64] Y. Berger, A. Wool, and A. Yeredor, "Dictionary attacks using keyboard acoustic emanations," in *Proceedings of the 13th ACM conference on Computer and communications security*, ACM, 2006, pp. 245–254.

[65] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," in *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, 2010, pp. 307–322.

[66] A. Shamir and E. Tromer, "Acoustic cryptanalysis: On nosy people and noisy machines," *Online at http://people. csail. mit. edu/tromer/acoustic*, 2004.

[67] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, "Temperature attacks," *IEEE Security & Privacy*, vol. 7, no. 2, pp. 79–82, 2009.

[68] M. Hutter and J.-M. Schmidt, "The temperature side channel and heating fault attacks," in *International Conference on Smart Card Research and Advanced Applications*, Springer, 2013, pp. 219–235.

[69] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs," in *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2014.

[70] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi, "Cryptanalysis of block ciphers implemented on computers with cache," in *Proceedings of the International Symposium on Information Theory and its Applications*, 2002, pp. 803–806.

[71] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, ACM, 2007, pp. 494–505, ISBN: 978-1-59593-706-3.

[72] E Bangerter, D Gullasch, and S Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *Proceedings of IEEE Symposium on Security and Privacy*, 2011.

[73] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2014, pp. 299–319.

[74] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *International workshop on cryptographic hardware and embedded systems*, Springer, 2001, pp. 251–261.

[75] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The em side—channel (s)," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2002, pp. 29–45.

[76] R. Rutledge, S. Park, H. Khan, A. Orso, M. Prvulovic, and A. Zajic, "Zero-overhead path prediction with progressive symbolic execution," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 234–245.

[77] Y. Han, I. Christoudis, K. I. Diamantaras, S. Zonouz, and A. Petropulu, "Side-channel-based code-execution monitoring systems: A survey," *IEEE Signal Processing Magazine*, vol. 36, no. 2, pp. 22–35, 2019.

[78] N. Sehatbakhsh, M. Alam, A. Nazari, A. Zajic, and M. Prvulovic, "Syndrome: Spectral analysis for anomaly detection on medical iot and embedded devices," in *2018 IEEE international symposium on hardware oriented security and trust (HOST)*, IEEE, 2018, pp. 1–8.

[79] N. Sehatbakhsh, A. Nazari, M. Alam, F. Werner, Y. Zhu, A. Zajic, and M. Prvulovic, "Remote: Robust external malware detection framework by using electromagnetic signals," *IEEE Transactions on Computers*, 2019.

[80] D. Spatz, D. Smarra, and I. Ternovskiy, "A review of anomaly detection techniques leveraging side-channel emissions," in *Cyber Sensing 2019*, International Society for Optics and Photonics, vol. 11011, 2019, 110110E.

[81] M. Dey, A. Nazari, A. Zajic, and M. Prvulovic, "Emprof: Memory profiling via em-emanation in iot and hand-held devices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 881–893.

[82] N. Sehatbakhsh, A. Nazari, H. Khan, A. Zajic, and M. Prvulovic, "Emma: Hardware/software attestation framework for embedded systems using electromagnetic signals," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 983–995.

[83] K. Sakiyama, M. Kasuya, T. Machida, A. Matsubara, Y. Kuai, Y.-i. Hayashi, T. Mizuki, N. Miura, and M. Nagata, "Physical authentication using side-channel information," in *2016 4th International Conference on Information and Communication Technology (ICoICT)*, IEEE, 2016, pp. 1–6.

[84] L. N. Nguyen, C.-L. Cheng, M. Prvulovic, and A. Zajić, "Creating a backscattering side channel to enable detection of dormant hardware trojans," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 27, no. 7, pp. 1561–1574, 2019.

[85] L. N. Nguyen, "Hardware trojan detection using the backscattering side channel," 2020.

[86] L. N. Nguyen, C.-L. Cheng, F. T. Werner, M. Prvulovic, and A. Zajic, "A comparison of backscattering, em, and power side-channels and their performance in detecting software and hardware intrusions," *Journal of Hardware and Systems Security*, pp. 1–16, 2020.

[87] L. N. Nguyen, B. B. Yilmaz, C.-L. Cheng, M. Prvulovic, and A. Zajić, "A novel clustering technique using backscattering side channel for counterfeit ic detection," in *Cyber Sensing 2020*, International Society for Optics and Photonics, vol. 11417, 2020, p. 1 141 709.

[88] R. Callan, A. Zajić, and M. Prvulovic, "Fase: Finding amplitude-modulated side-channel emanations," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 43, 2015, pp. 592–603.

[89] R. Callan, A. Zajic, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, IEEE, 2014, pp. 242–254.

[90] F. T. Werner, A. R. Djordjević, D. I. Olćan, M. Prvulovic, and A. Zajić, "Experimental validation of localization method for finding magnetic sources on iot devices," in *2018 International Symposium on Electromagnetic Compatibility (EMC EUROPE)*, IEEE, 2018, pp. 413–418.

[91] A. Zajic and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 56, no. 4, pp. 885–893, 2014.

[92] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[93] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do, "Software-artifact infrastructure repository," *UR L http://sir. unl. edu/portal*, 2006.

[94] N. Andronio, S. Zanero, and F. Maggi, "Heldroid: Dissecting and detecting mobile ransomware," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2015, pp. 382–404.

[95] Ettus, *Usrp-b200mini*, `https://www.ettus.com/product/details/USRP-B200mini-i`, accessed February 4, 2018.

[96] *Open syringe-pump source code and project*, `https://github.com/naroom/OpenSyringePump`, Last Accessed: 2018-08-01.

[97] *Pid controller soldering iron code and project*, `https://github.com/sfrwmaker/soldering_controller`, Last Accessed: 2018-08-01.

[98] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero, "An experimental security analysis of an industrial robot controller," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 268–286.

[99] *Robotic arm code and project*, `https://lifehacker.com/build-a-kickass-robot-arm-the-perfect-arduino-project-1700643747`, Last Accessed: 2018-08-01.

[100] *Arduino servo refrence library*, https://www.arduino.cc/en/Reference/Servo, Last Accessed: 2018-08-01.

[101] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[102] L. Deng, D. Yu, *et al.*, "Deep learning: Methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[103] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems*, vol. 1, no. 4, pp. 111–122, 2011.

[104] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, p. 533, 1986.

[105] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[106] M. D. Zeiler, M Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, *et al.*, "On rectified linear units for speech processing," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, IEEE, 2013, pp. 3517–3521.

[107] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[108] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, Springer, 2010, pp. 177–186.

[109] P. Juyal, S. Adibelli, N. Sehatbakhsh, and A. Zajic, "A directive antenna based on conducting discs for detecting unintentional em emissions at large distances," *IEEE Transactions on Antennas and Propagation*, pp. 1–1, 2018.

[110] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative

embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, IEEE, 2001, pp. 3–14.

[111] B. Wijnen, E. J. Hunt, G. C. Anzalone, and J. M. Pearce, "Open-source syringe pump library," *PloS one*, vol. 9, no. 9, e107216, 2014.

[112] T. Abera, N Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: Control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 743–754.

[113] B. B. Yilmaz, A. Zajić, and M. Prvulovic, "Modelling jitter in wireless channel created by processor-memory activity," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2018, pp. 2037–2041.

[114] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[115] Olimex, *A13-olinuxino-micro user manual*, `https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware`, accessed April 3, 2016.

[116] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan*, 2005.

[117] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom, *Sliding right into disaster: Left-to-right sliding windows leak*, Conference on Cryptographic Hardware and Embedded Systems (CHES) 2017, 2017.

[118] ARM, *Arm cortex a8 processor manual*, `https://www.arm.com/products/processors/cortex-a/cortex-a8.php`, accessed April 3, 2016.

[119] Keysight, *N9020a mxa spectrum analyzer*, `https://www.keysight.com/en/pdx-x202266-pn-N9020A/mxa-signal-analyzer-10-hz-to-265-ghz?cc=US&lc=eng`, accessed February 4, 2018.

VITA

Haider Adnan Khan grew up in Dhaka, Bangladesh, and received a B.Sc. degree in Electrical and Electronic Engineering from Bangladesh University of Engineering and Technology, Bangladesh in 2006, and an M.Sc. degree in Electrical Engineering and Information Technology from Karlsruhe Institute of Technology, Germany in 2011.

He joined Georgia Institute of Technology in Fall 2015 and pursued Ph.D. degree in School of Electrical and Computer Engineering. He has worked as a Graduate Research Assistant in the Electromagnetic Measurements in Communications and Computing Lab focusing on electromagnetic side-channel information leakage for non-adversarial program execution monitoring. His research interests span areas of machine/deep learning, signal/image processing, and side-channel analysis.